PROVIDING FAULT TOLERANCE FOR TRANSACTIONAL WEB SERVICES

by

IVAN VASQUEZ

(Under the Direction of John A. Miller)

ABSTRACT

Web service-based transactional business processes require a high degree of reliability in order to guarantee consistent results. In case of failure, participants must take the necessary actions to leave the process in a globally-correct state. Conventional Web services, however, frequently lack essential reliability features. Based on the Web Services Transactions specifications, we present a framework that provides fault tolerance to the services involved in a transactional business process, leveraging techniques in logical logging and application recovery. In contrast to current implementations, it aims to minimize the impact on existing applications with regards to performance and code changes.

INDEX WORDS: Web services, Transactions, Business Processes, Workflow, Fault Tolerance, Recovery, Logical Logging

PROVIDING FAULT TOLERANCE FOR TRANSACTIONAL WEB SERVICES

by

IVAN VASQUEZ

B.S., Universidad ICESI, Cali, Colombia, 1999

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2006

PROVIDING FAULT TOLERANCE FOR TRANSACTIONAL WEB SERVICES

by

IVAN VASQUEZ

DEDICATION

To my wife Carolina and my parents, Mary and Guillermo

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Dr. John A. Miller for his support and great encouragement to research. Special thanks to Dr. Amit Sheth for offering his experience and expert advice. My appreciation also goes to my friends Kunal, Karthik and Meena, who helped me during these years at the LSDIS Lab.

TABLE OF CONTENTS

APPENDIX

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Transactional workflow systems have been available for long time. However, their elevated interoperability costs have kept them limited to mission-critical enterprise systems. Thanks to Service Oriented Computing and in particular, Web services, the popularity of such systems promises to increase in the near future: XML is now a widely accepted standard to represent exchanged data, applications can easily expose select business logic as Web service operations, while the Internet provides an affordable yet pervasive communication link.

Nowadays, emerging languages like WS-BPEL (BPEL) [18] allow existing Web services to be composed into business processes. Because it is often desirable for these to arrive to a commonly agreed outcome, the Web Services Transactions specifications [16] [17] [19] formalize the requirements to support transactions and other coordinated activities. Unfortunately, since conventional services are usually unable to guarantee their operational stability, the resulting transactional processes cannot be deemed reliable or necessarily consistent. Therefore, in order to guarantee transactional process integrity, compliant implementations must deliver the fault tolerance requirements needed in a heterogeneous environment of autonomous participants.

Based on our analysis of the requirements to deliver reliable Web services in the context of transactional processes, we present OpenWS-Transaction, a framework that leverages logical logging and forward recovery to enable existing services to meet their reliability requirements. A prototype implementation shows how this framework can be integrated into existing services by introducing minimal changes to their application code. The results of our experimentation demonstrate that fault tolerance can be delivered without causing significant performance overhead. This work has been implemented as part of the METEOR-S project, which deals with adding semantics to the complete lifecycle of Web services and processes [37].

The Web services targeted by our study are commonly the result of existing application business logic. As such, they generally access an underlying database management system (DBMS) which, despite providing local data consistency, is insufficient to guarantee the integrity of the distributed business processes they collaborate with. In the presence of operational failures, they may loose all or part of their state, leaving parts of the process in an unknown status, which leads to process inconsistency. Hence, our approach to fault tolerance is primarily interested in allowing a running transactional process to resume normal execution past the point of failure [11], instead of resorting to complete rollback (i.e., backward recovery) or to take all the necessary actions to reach an acceptable state (i.e., forward recovery) [12].

CHAPTER 2

PROVIDING FAULT TOLERANCE FOR TRANSACTIONAL WEB SERVICES[1]

[1] Ivan Vasquez, John Miller, Kunal Verma. Submitted to *International Journal of Web Services Research*

1. Introduction

Transactional workflow systems have been available for long time. However, their elevated costs (associated with proprietary middleware, specialized applications and private communication channels) have kept them limited to mission-critical enterprise systems. Thanks to the increased interoperability facilitated by Service Oriented Computing and in particular, Web services, the popularity of such systems promises to increase in the near future: There is now a standard way to represent exchanged data, applications can easily expose select business logic as service operations while the Internet provides an affordable yet pervasive communication link.

Nowadays, emerging languages like WS-BPEL (BPEL) [18] allow existing Web services to be composed into business processes. Because it is often desirable for these to arrive to a commonly agreed outcome, the Web Services Transactions specifications [16] [17] [19] formalize the requirements to support transactions and other coordinated activities. Unfortunately, since conventional services are usually unable to guarantee their operational stability, the resulting transactional processes cannot be deemed reliable or necessarily consistent. Therefore, in order to guarantee transactional process integrity, compliant implementations must deliver the fault tolerance requirements needed in a heterogeneous environment of autonomous participants.

In this work, we analyze the requirements to deliver reliable Web services in transactional business processes based on the aforementioned specifications, which enjoy growing acceptance. It presents a framework that leverages logical logging and partial forward recovery to enable existing services to meet their reliability requirements. A prototype implementation shows how the proposed framework can be integrated into existing services by introducing minimal changes to their application code. The results of our experimentation demonstrate that fault tolerance can be delivered without causing significant performance overhead.

We use the term *conventional Web services* to refer to the Web services targeted by our study, which are commonly the product of externalized application business logic. As such, they generally access an underlying database management system (DBMS) which, despite providing local data consistency, is insufficient to guarantee the integrity of the distributed business processes they collaborate with. In the presence of operational failures (e.g., site failures, network partitions, database downtime), they may loose all or part of their state, leaving parts of the process in an unknown status, which leads to process inconsistency. Our approach to fault tolerance is primarily interested in allowing a running transactional process to resume normal execution past the point of failure [11], instead of resorting to complete rollback (i.e., backward recovery) or to take all the necessary actions to reach an acceptable state (i.e., forward recovery) [12].

The next section starts with an overview of related Web service specifications. Section 3 describes scenarios where fault tolerance becomes necessary. Section 4 explains the framework's architecture. Sections 5 and 6 delve into the details of when and how logging and recovery should be used to support service-based transactional business processes. Section 7 refers to related work on Web service composition, workflow systems, transactions and logging, which this paper elaborates upon. Prototype implementation details are given in Section 8 and the results of our evaluation are described in Section 9. The paper concludes with a discussion on how the proposed framework satisfies the requirements and discusses issues that are the subject of future work.

2. Background

The issues that our work intends to address have their origins in the areas of transactions and workflow systems, which have been subject of extensive research in the past. Workflows are activities involving the orderly execution of multiple tasks by different entities (whether computational or human) [27], while *transactional* workflows [1] [7] [32] are a special variation of the former that access heterogeneous, autonomous and distributed systems, and support the selective use transactional properties.

With the increasing popularity of the Service Oriented Architecture (SOA) and Web services, BPEL has emerged as a language that allows one to model business processes in terms of interactions among services. However, in absence of transaction management specifications, it would be the responsibility of the process designer to ensure commonly agreed upon outcomes in transactional workflows. Such a requirement would have to be addressed reiteratively and, due to its complexity, would be inappropriate for orchestration languages such as BPEL [18]. Instead, enforcement of transactional properties is meant to be delegated to specialized entities and protocols defined by the following specifications:

- WS-Coordination (WS-C) defines a *coordination context* used to share information about activities carried across multiple services, as well as *coordination* and *registration* services with operations for creating such activities and registering their *participants*. WS-C is extensible, becoming an integral part of new *coordination types*. In particular, the Web Services Transactions specifications (described below) rely on this specification by defining coordination types suitable for short and long running transactions.
- WS-AtomicTransaction (WS-AT) targets existing transactional systems with short interactions and full ACID properties. To guarantee *atomicity*, the transaction protocols available to WS-AT are derivatives of the Two-Phase Commit (2PC) protocol, characterized by all-or-nothing results. In order to guarantee *consistency* and *isolation*, WS-AT transactions follow the closed nested model [22]: The effects of individual operations (i.e., subtransactions) are not readily visible to others; instead, participants leave active, uncommitted transactions in their local databases until the outcome of the process is decided. Consistency also depends on the isolation level [4] at which participants interact with concurrent transactions in their local database systems. Finally, the *durability* property is collectively provided by the underlying database systems.
- WS-BusinessActivity (WS-BA), on the other hand, is intended for applications involving business processes of long duration. Its *AtomicOutcome* coordination type dictates that all participants must either confirm or cancel their work, whereas the *MixedOutcome* coordination type is suitable for transactions whose final outcome may not be solely decided by the unanimous vote of all participants. In WS-BA, transactions follow the open nested model [22]: Preliminary operations are immediately persisted and visible to others, but *compensating transactions* are needed to undo their effects [5]. Variations of its BusinessActivity protocol [17] support this behavior, which increases concurrency and suits a wider range of applications, including those involving asynchronous and human interaction.

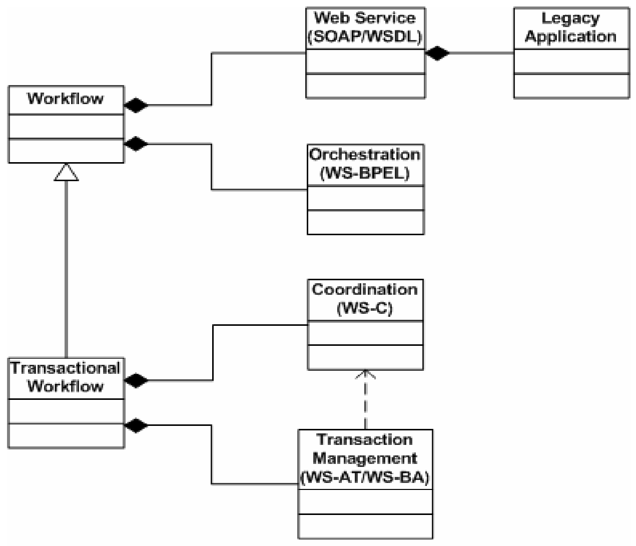


Figure 1: Relationships among the intervening Web services specifications

As suggested by [34], complementing BPEL with the aforementioned specifications enables the creation of workflows that, besides implementing the business logic, meet the necessary functional requirements to integrate heterogeneous applications. Figure 1 shows the conceptual model upon which transactional workflows can be built: At the core of any Web service-based workflow system are specifications for message communication (SOAP), service description (WSDL) and orchestration (BPEL). Additionally, transactional workflows require specifications for activity coordination (WS-C) and transaction management (WS-AT and WS-BA). In line with the composable architecture of Web services, these specifications leverage the infrastructure provided by WS-Addressing (which provides a standard means to specify service endpoint addresses), WS-Policy (used to specify constraints that must be met in order to invoke service operations), and WS-ReliableMessaging (which provides higher guarantees about message delivery), to cite just a few.

3. Motivation

Consider a possible scenario for a transactional business process: Once the transaction begins, all intervening participants complete their work and report their outcome to the coordinator. After gathering all votes, the coordinator decides the transaction’s final outcome and, assuming it is affirmative, proceeds to commit every participant. In the meantime, a site failure causes an uncommitted participant to crash as illustrated in Figure 2. Since the outcome has already been determined and some participants have already committed, there is no way back: The failed participant has also the obligation to commit. However, the crash caused volatile state information to vanish and local transactions were implicitly rolled back by the underlying database. Because most Web services have no way to recover their original state, the coordinator would not be able to properly finish this transaction, leaving it in an unknown status.

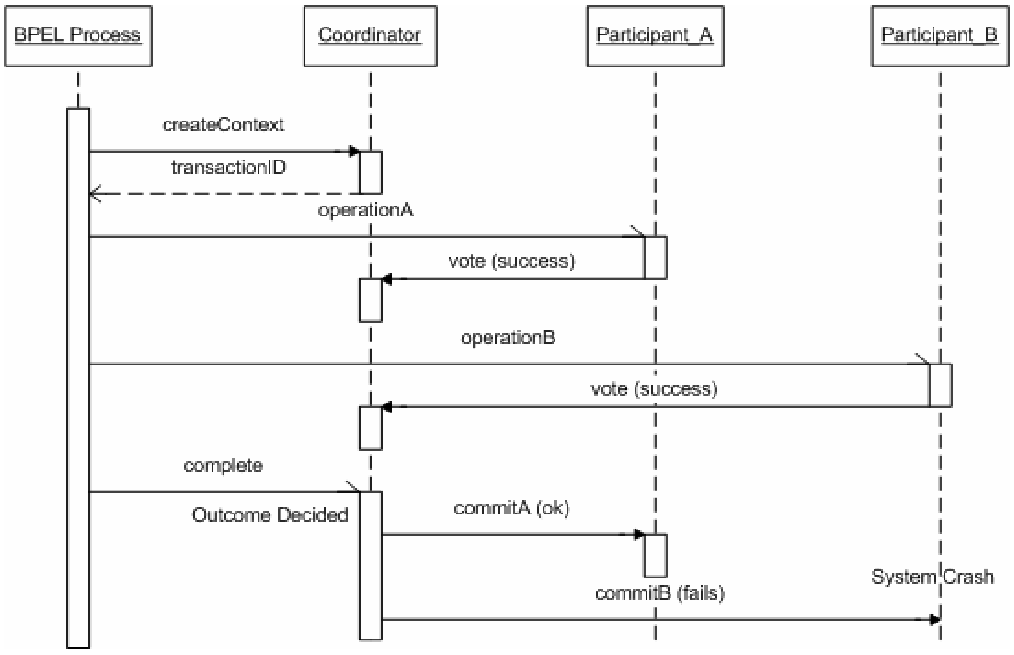


Figure 2: An unsuccessful transactional process where one of its participants crashed

Under these circumstances, several things become apparent: In order to resume process execution, it is necessary to restore vital system information such as the transaction identifier and coordination type. It may also be necessary to re-execute pending local database transactions to reconstruct the state of participants at the time of failure. Likewise, participants in a long-running transaction need a failure recovery mechanism that allows recovering their state. Only in that way will they be able to compensate the effects of tentative operations as directed by the coordinator.

In another scenario, the coordinator itself could become unavailable in the middle of a process, leaving pending operations at multiple participants. Eventually, every unfinished process would stall waiting for its intervention. Once its hosting site becomes operational, the coordinator must be able to restore the state of all transactions it manages; otherwise, individual operations may be left in a globally-incoherent status.

In an attempt to address the above scenarios, our work assumes (1) that it is possible to introduce minimal changes to existing services, (2) that failed sites are the result of temporary failures (as opposed to disastrous situations, in which they become permanently unavailable), and (3) that operational failures of the process orchestration engine are corrected by its own recovery mechanism[2].

4. Architecture

To deliver transactional properties to a process (or sections of it), users begin by defining a *transactional scope* (also called *coordination scope* [34]), which is implicitly created by enclosing relevant activities within calls to the coordinator, as shown in Figure 3. All activities contained within the scope are guaranteed to complete according to the selected coordination type (i.e., WS-AT, WS-BA AtomicOutcome, WS-BA MixedOutcome) and any number of scopes can be defined within a single process.

[2] To date, several BPEL engines (including ActiveBPEL, an open source implementation) feature process recovery.

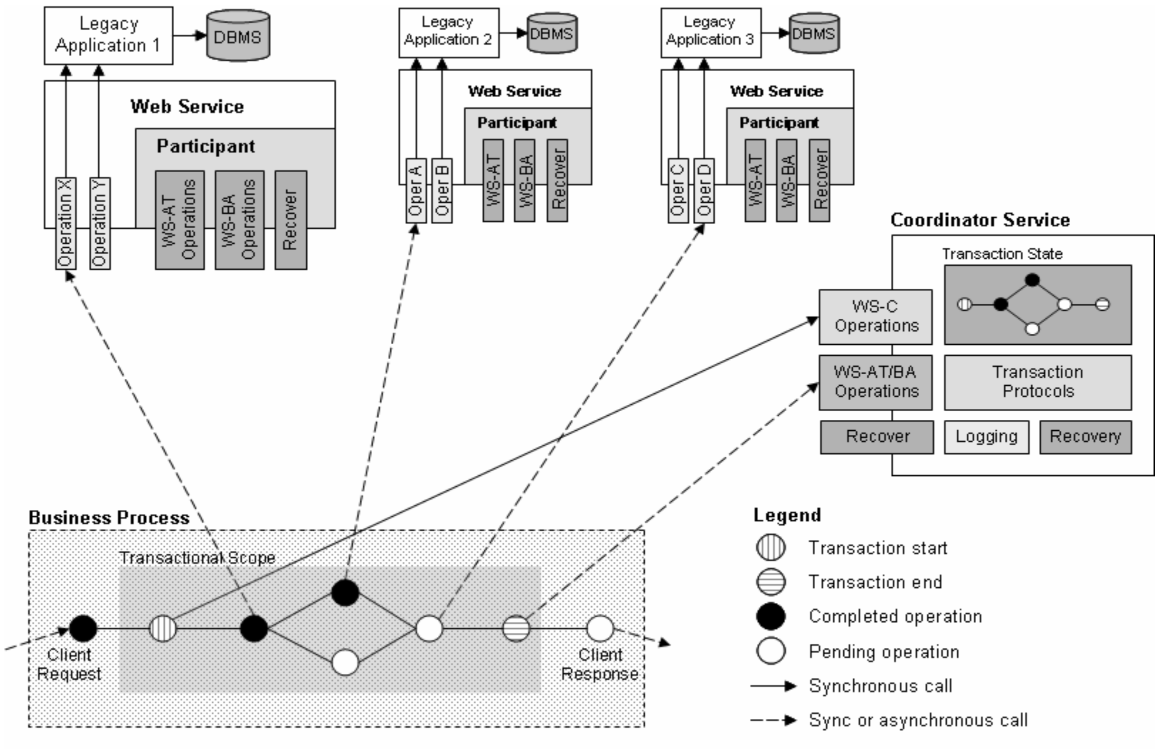


Figure 3: Entities and some of their interactions in a transactional business process

In its basic form, a transactional scope results into a two-level nested transaction [22] composed of one subtransaction for each Web service operation. Nevertheless, subtransactions can also correspond to entire transactional workflows, forming specific types of complex transactions [1] [33]. The present work focuses on providing support to the former kind of transactional processes.

Since BPEL can be used to compose arbitrarily complex processes, it is the responsibility of the process designer to ensure that the structure of service interactions (e.g., looping instructions) and the affected data items do not lead to data contention. Otherwise, an atomic transactional process can effectively deadlock as a result of these interactions with the underlying databases. It is important to note that when a participant performs multiple operations on behalf of a transaction, from the perspective of the underlying database, each operation corresponds to a separate and unrelated transaction, and therefore simultaneous access to the same data will be blocked under atomic transaction coordination.

Central to any implementation of Web service-based transactions are the coordinator and protocol-aware participant services, whose structure, features and interaction are described in the following sections.

4.1. Coordinator

In our prototype, the coordination and registration services defined by WS-C are collectively represented by the coordinator service, which is responsible for creating new transactions and for activating their participants. Given its central role, it is also best suited for acting as a transaction manager, executing the transaction protocols defined in WS-AT and WS-BA (i.e., 2PC[3], Completion and BusinessAgreement). To support failure recovery, coordinators are provided with a logging facility used to record critical events throughout the transaction's lifespan (logging is explained in detail in Section 5).

This aggregate design consolidates the deployment of several closely-related services while simplifying the framework's architecture. Despite its multiple responsibilities, the coordinator is as capable as the individual services it replaces. The resulting service is outlined in Figure 4.

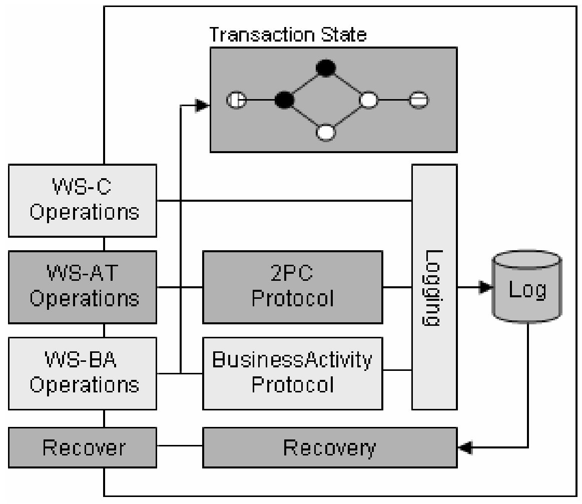


Figure 4: Coordinator Service

4.2. Participants

Frequently, transaction-oriented Web services result from legacy applications that expose select functionality to business partners. These applications usually access a database system back end, as illustrated in Figure 5. In order for these services to participate in a distributed transaction, they must adopt coordination protocols and provide minimum guarantees about their reliability. With our framework, they can do so by taking advantage of specific features provided by the *Participant* entity (also referred to as *Coordination Participant* [34]). Among its key features is the ability to intercept and record protocol state changes and database activity to support recovery. This way, conventional services not only become aware of the coordination infrastructure, but also gain fault resilience. In addition to preexisting operations, services are augmented with protocol-specific features, as outlined in Figure 6.

Previous work on Web service transactions [20] has suggested the use of intermediary services (proxies) for protocol adaptation, playing an analogous role to that of our *Participant* entity. While the use of proxies is in some cases an adequate solution (e.g., in dynamic service discovery), we believe that supporting a transaction-oriented service inevitably requires direct application involvement (e.g., to commit an atomic transaction or to compensate a business activity) and therefore a simpler and more efficient approach consists in adding extensions to the service itself. Moreover, while the proposed proxies require customization to match the operations available in a given application, our framework takes the opposite approach: By delivering only protocol operations, it is possible to provide application-independent transactional support.

[3] For simplicity, our prototype implements *centralized 2PC*: Participants communicate only with the coordinator, not among themselves.

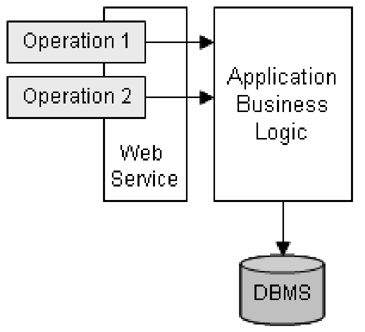


Figure 5: Conventional Web service

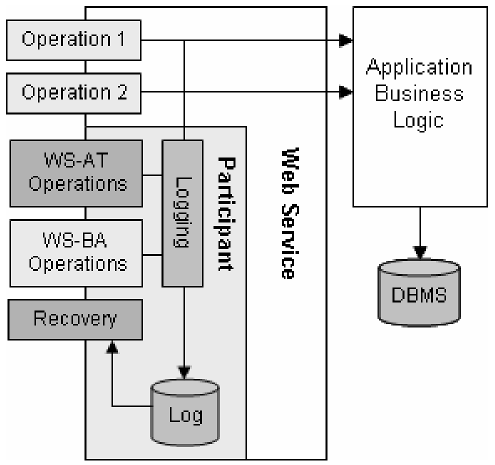


Figure 6: Coordinated participant Web service

4.3. Interaction

The interactions among the intervening entities are, to a great extent, dictated by the reference specifications. A transaction begins when a running business process enters into a transactional scope. There, the *createCoordinationContext* operation signals the coordinator service to create a new coordination context, which differentiates a transaction from others by means of a unique identifier and contains essential information such as its coordination type, status and expiration time. As the process executes activities in the transactional scope, invoked participants first call the coordinator's *register* operation to request their inclusion in the transaction. The coordinator then records their endpoint URL on persistent storage for future reference.

Following registration, a participant performs all relevant business operations and records their outcome using the locally-available logging facility. Once the transactional scope reaches its end, the coordinator and all participants handle the transaction according to the active coordination type, as follows:

Atomic Transactions (WS-AT)

- Using the standard 2PC protocol, the coordinator first invokes every participant's *prepare* operation to request their votes with regards to the transaction. They respond by calling back on either its *prepared* operation, to notify the intention to commit the transaction, *aborted*, to notify the intention to roll it back, or *readOnly* to notify that their work had no effect on the transaction. In our framework, the *prepare* operation is also used during coordinator recovery, when participants are asked to resubmit their votes.
- Alternatively, using an optimization known as *unsolicited vote* [30], participants can submit their votes to the coordinator as soon as their part of the work is completed.

The second option saves one message for every registered participant, as the outbound *prepare* message from the coordinator to participants would be unnecessary. However, since the coordinator cannot possibly determine whether any participants remain to vote on the sole basis of unsolicited votes, the initiating process must notify the coordinator about the end of the transactional scope.

To notify the coordinator about the end of the transactional scope, the initiating process uses the *completion protocol* [16]. At the coordinator, this event marks the beginning of the transaction protocol, which starts by gathering all votes (if standard 2PC is used) or by simply evaluating all votes already submitted (if unsolicited vote is used). Then, the coordinator determines the transaction's outcome and records it immediately on stable storage.

Following this determination, the coordinator directs participants to finalize any tentative operations. In case of an affirmative outcome, the *commit* operation causes all associated local database transactions to be committed, whereas *rollback* causes them to be aborted. Upon completion, the participant logs this event, responds to the coordinator with either *committed* or *aborted* as necessary, and forgets about the state of transaction (however, information in the log is still preserved). Once all participants acknowledge the final decision, the coordinator records this event in its log, notifies the initiating business process and also forgets about the state of the transaction.

Business Activities (WS-BA)

Though conceptually similar, the BusinessAgreement protocol allows more flexible interactions than its WS-AT counterpart. In it, participants notify the coordinator about their operations by invoking *completed*, used upon completion of all requested activities, *exit*, to request their exclusion from further transaction processing, or *fault*, to signal an exceptional condition. This scheme contemplates the possibility of processes whose final outcome may not be decided by unanimous votes.

Unlike atomic transactions, the long-running nature of business activities allows coordinators to request the cancellation of ongoing work (e.g., an order that is being fulfilled but has not been shipped). Hence, participants include a *cancel* operation, while coordinators provide the corresponding *cancelled* acknowledgement operation.

The coordinator's final decision depends on the selected coordination type. Under AtomicOutcome, just as in 2PC, all participants must be directed to either *compensate*, undoing the effects of their operations, or *close*, leaving their operations as they were completed (the implications of compensation are discussed in Section 5.4.4.) Under MixedOutcome, however, any combination of final notifications is permissible. Once participants receive the final command, they acknowledge by responding with either *compensated*, to notify a successful compensating action, *closed*, to notify their normal finalization, or *fault*, to signal an exception while applying compensation.

For coordinators and participants, the framework defines an additional *recover* operation that allows external entities (i.e., the coordinator in the case of participants, a master coordinator in the case of coordinators) to trigger their recovery process, bringing these services back to the state immediately previous to an operational failure. Recovery alternatives are explained in Section 6.

Transaction processing may encounter a number of exceptional conditions which prevent their normal development. The following is a non-exhaustive list of protocol exceptions caught by the current prototype implementation:

- *InvalidCoordinationType:* An invalid coordination type was requested at context creation.
- *NonExistingContext:* A participant tried to register (or vote) an operation, but the coordinator does not know about such transaction.
- *AlreadyRegistered:* A participant wants to be included in a transaction, but according to the coordinator, it has already done so.
- *InconsistentInternalState:* A participant has reached a state from which the transaction cannot be automatically recovered. Transaction status is changed to in-doubt and requires user intervention to be corrected.

Table 1: Coordinator operations by defining specification

Specified in	*Operation(s)*	*Description*
WS-C	CreateCoordinationContext	Begins a new transaction by creating a coordination context.
	Register	Activates a participant in the transaction.
WS-AT	Commit (completion protocol)	Used by applications to notify the coordinator about their intention to commit the transaction.
	Rollback (completion protocol)	Used by applications to notify the coordinator about their intention to roll back the transaction.
	Prepared	Notifies the coordinator about the participant’s ability and willingness to commit.
	ReadOnly	Notifies the coordinator about a participant operation that did not change the state of data.
	Aborted	Notifies the coordinator about the participant’s intention to roll back the transaction.
	Committed	Acknowledges the coordinator about a participant’s successful local database commit.
	Replay	Used by participants after a recoverable failure to ask the coordinator to re-send the last message.
WS-BA	Completed (participant completion)	Used by participants to notify the coordinator about the completion of tasks.
	Exit	Notifies the coordinator of the participant’s intention to be excluded from further processing.
	Fault	Notifies the coordinator of the participant’s exceptional condition.
	Compensated	Notifies the coordinator about the success in applying compensating actions to previous work.
	Canceled	Acknowledgment from participants about the coordinator’s intention to cancel ongoing work.
	Closed	Acknowledgment from participants about the coordinator’s intention to finish their work.
Prototype	Recover	Used by external entities to initiate the recovery process on the coordinator.

Table 2: Participant operations by defining specification

Specified in	*Operation(s)*	*Description*
WS-AT	Commit	Directs the participant to commit its work.
	Rollback	Directs the participant to abort its work.
	Prepare	Used by coordinators to gather votes from participants.
WS-BA	Complete (coordinator completion)	Used by coordinators to notify the participant about the completion of tasks.
	Compensate	Directs the participant to compensate its work.
	Close	Directs the participant to leave its work as completed.
	Cancel	Directs the participant to cancel an ongoing task.
	Exited	Acknowledges the participant about an earlier request to be excluded from the transaction.
	Faulted	Acknowledges the participant about an earlier notification of an exceptional condition.
Prototype	Recover	Used by external entities to initiate the recovery process on the participant.

Tables 1 and 2 summarize the operations available to coordinators and participants in accordance to the reference specifications.

In summary, the proposed architecture is largely influenced by its reference specifications. At the same time, it attempts to simplify the framework's usability and to minimize application overhead (Section 9 presents an evaluation of performance).

5. Transaction Logging

Transactions have always relied on logging and Web service-based transactions are no exception. While in centralized transactional systems logging is used to enforce *data consistency*, in distributed environments it can also help to guarantee the *consistent execution of distributed protocols* [40]. The importance of this role can be realized considering that a distributed protocol for participants that cannot recover from failures has the potential of leaving the system in an incoherent state. By leveraging existing techniques, logging provides a protocol event history that enables participant recovery, allowing the protocol to be continued exactly at the state it was interrupted by a failure.

Since conventional Web services rarely provide the necessary fault-tolerance guarantees to take part in a distributed transaction, the reasons for using logging are twofold: The need to deliver *application state recovery*, represented in terms of vital system variables (which collectively form the *coordination context*), and the need to provide *transaction protocol recovery* in order to guarantee its correct execution. While this discussion focuses on supporting the 2PC protocol, logging can be used just as effectively with other transaction protocols [26].

Briefly explained, transactional process logging consists in having coordinators and participants record specific events as the transaction develops. The resulting log records represent key milestones in the transaction's lifetime, which can later be used to resume its execution from the point of failure [40].

5.1. Logging Architecture

Our prototype implementation uses a lightweight embedded database system as a logging device. Such a device is present at every coordinator and participant site that uses the framework, and is analogous to the concepts of *Global Persistent Store* and *Local Persistent Store* defined in [41]. Issues such as atomic log operations, physical log file formats, concurrent file access and buffer management are all delegated to this mechanism, which guarantees transactional properties by itself.

5.1.1. Coordinator Log

Fault-tolerant coordinators are responsible for recording on stable storage every transaction, its status, coordination type and registered participants. Figure 7 and the following sections describe the coordinator's logging schema. In the following diagrams, *PK* and *FK* denote primary and foreign keys, respectively.

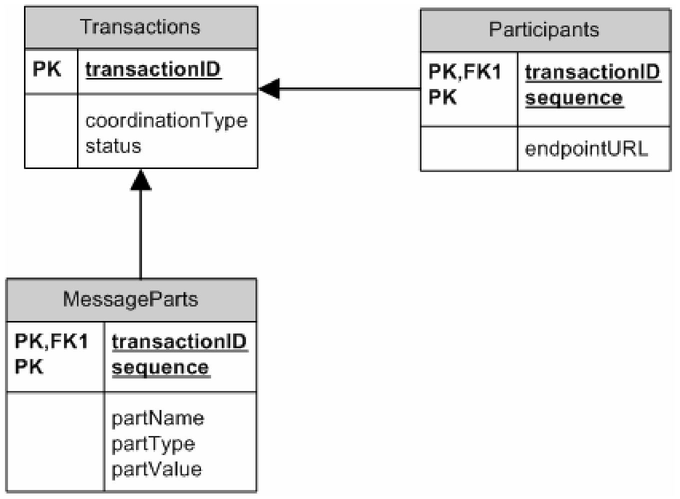


Figure 7: Coordinator logging schema

The *transactions* relation stores a tuple for every transaction managed by the coordinator. It records the unique transaction identifier assigned to the coordination context, the coordination type, which defines transaction behavior, and the transaction's status, which can take any of the following values:

Table 3: Possible transaction states

Transaction Status	Coordinator Action
active	Waiting for participants to perform their all operations
preparing	Evaluating participant votes before a decision is made
committing	Completing the transaction protocol by committing all participants
aborting	Completing the transaction protocol by aborting successful participants
ended	Transaction has finished (regardless of outcome)
recovering	Recovering from a site failure; prevents fulfillment of new requests
in-doubt	An exceptional condition kept the coordinator from ending the transaction. User intervention is required

For every registered participant, the *participants* relation stores the endpoint URL where the service can be reached, allowing the coordinator to contact active participants during crash recovery. Records in this relation are uniquely identified by combining the transaction's unique identifier and the ordinal sequence in which the participant was registered.

Optionally, coordinators can also keep a copy of the message that triggered the business process in case it has to be restarted. Each tuple in the *messageParts* relation represents a part of the startup message: its name, data type and value.

Information about operation outcomes is only stored at participants. Operation outcomes are not logged when reported to the coordinator; this reduces logging overhead and avoids potential inconsistencies that could arise from keeping duplicate information at multiple sites. Instead, coordinators keep an in-memory image of transaction state, which can be rebuilt by polling registered participants.

By collecting the above information, coordinators are capable of identifying active transactions at the time of failure, contacting their participants and gathering votes, regaining control over the system. A detailed description of event logging is given on Section 5.2.

5.1.2. Participant Log

Participant transaction logging is more involved than that of coordinators. In addition to recovering the local copy of the coordination context, depending on the coordination type, participants are also responsible for guaranteeing the stability of tentative operations (WS-AT) and for providing means to compensate the effects of others (WS-BA). Figure 8 illustrates the logging schema present at every participant; the following sections describe its usage depending on the coordination type.

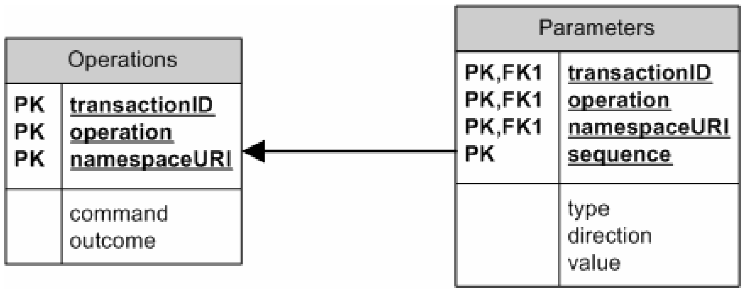


Figure 8: Participant logging schema

Atomic Transactions (WS-AT)
As described in Section 2, transactions under this coordination type are of short duration and guarantee ACID properties. While this allows a more consistent set of recovery scenarios than WS-BA, the implication is that transactions must be performed in full isolation, following the *closed nested* transaction model [22]. Normally, this forces participants to create and leave uncommitted database transactions, locking affected data items until a definitive decision is received from the coordinator. Thus, the transaction protocols available to WS-AT guarantee complete data consistency at the cost of reduced concurrency.

In order to cope with failures, participants must provide *pre-commit* behavior: Once an operation is invoked and its positive vote is communicated, the participant has tacitly promised its ability to finalize the operation at a later time. A frequent problem, however, is that most commercially-available database systems either do not provide this behavior or provide it only as part of separate programming interfaces [4] (e.g., as part of an XA interface [42]). Instead, the default action in case of site failures is to undo all uncommitted operations [6]. Therefore, fault tolerant services must also provide a mechanism to restore provisional database changes present as of the time of failure.

This motivated us to shift the task of database activity monitoring to the application level, becoming an integral part of the framework and enabling participants to replay data operations without relying on support from their underlying databases. However, because neither application database schemas nor vendor-specific database syntaxes can be anticipated, it is imperative to log activity in a generic way. To this end, we combine a logging technique known as *logical logging* [15] with the generic syntax provided by the underlying data access API (e.g., JDBC or ADO.NET). Thanks to logical logging, rather than

[4] An exception is PostgreSQL v.8.1, which recently made available such a feature as part of its command interface.

tracing pre- and post- images of the data, only the operations performed against it are recorded, allowing a single logging schema to suit any common database interaction.

Regardless of the coordination type, the participant's *operations* relation records all operations executed on behalf of a transaction (including those resulting in a negative vote, read-only operations and protocol operations). As shown in Figure 8, this relation associates the transaction identifier (*transactionId*) with a unique identifier for the participant (*namespaceURI*), the operation's name (*operation*) and its outcome (*outcome*).

Additionally, for WS-AT, the *command* attribute records the operation performed on the underlying data, which corresponds to a database command. Command types can range from conventional SQL statements to procedure calls and parameterized statements as those shown in Table 4. Using the *parameters* relation, the framework can record and reproduce an arbitrary number of parameters associated with an operation. For each parameter (denoted by a question mark in Table 4), its value, ordinal position (*sequence*), data type (*type*) and direction (i.e., input, output or both) are captured.

Table 4: Sample logged SQL statements and procedural calls

Operation Type	**Example**
SQL Statement	`UPDATE flights SET reserved = ? WHERE source = ? AND destination = ?`
Procedure or function call	`{? = call reserve_flight(?, ?, ?)}`

Business Activities (WS-BA)
The long-running nature of this kind of transactions requires participants to persist individual database operations as soon as they are performed, even though they are logically tentative. This behavior agrees with the open-nested transaction model [22], in which the effects of subtransactions are readily visible to others. If this were not done, the concurrency penalty incurred by conventional data isolation could not be tolerated by most applications.

Since committed operations by definition cannot be rolled back, *compensating operations* are required to undo partial work. Compensation consists of *semantically* canceling the effects of an operation, as opposed to physically restoring data back to a previous state (i.e., by means of a rollback). Unfortunately, due to the effects of open-nested transactions, compensation may lead to the undesirable need to issue *cascading aborts* [10]: a chain of related compensating actions required to fix side effects created by the lack of isolation. Given these implications and the nature of some application domains, it is important to note that compensation may not always be a feasible option.

It is not our intention to provide guidelines on implementing compensating transactions. The mechanics of compensation are highly application-dependent [20], so it is the responsibility of the application designer to provide suitable compensating actions for each business operation. As with Sagas [5], applications coordinated with WS-BA must provide a compensating operation for every business operation so that, given the transaction's unique identifier, all necessary actions are taken to semantically undo preliminary work.

Given the previous facts, it is clear that in WS-BA database activity logging is unnecessary: From the perspective of the local database system, all operations are immediately committed, so tentative operations cannot be lost as a result of failures. Logging is solely dedicated to record protocol state transitions, using a subset of the schema described in Section 5.1.2.

5.2. Transaction Event Logging

Because logging occurs at the application level, it is especially important to ensure it does not adversely affect service performance. Two strategies commonly used to attain good logging performance are:

- Minimizing the number of logged events required to guarantee recovery.
- Reducing the number of forced writes to stable storage.

These optimizations can be put into practice (1) by limiting logging to only key state transitions in a process' lifecycle and (2) by postponing the actual writing of log records until others have to be written. Table 5 summarizes the sequence of events that must be logged to guarantee recoverability of Web service transactions from the perspective of WS-C and WS-AT. The rightmost column attempts to optimize logging by identifying conditions in which force-writing can be avoided without leading to an unrecoverable situation.

Table 5: Event logging requirements

	Service	Event	Source	Force Log?
1	Coordinator	Context Created	Process	No
2	Coordinator	Participant Registered	Participant	No
3a	Participant	Operation Completed (COMMIT)	Self	Yes (vote + activity)
3b	Participant	Operation Completed (ABORT/READONLY)	Self	Yes (vote)
4	Coordinator	Operation Voted	Participant	Not logged
5	Coordinator	Transactional Scope Ended	Process	Yes
6	Coordinator	Outcome Determined	Self	Yes
7a	Participant	Outcome Notified (COMMIT)	Coordinator	Yes (before & after)
7b	Participant	Outcome Notified (ABORT/READONLY)	Coordinator	Yes
8	Coordinator	All Acknowledged	Participant	Yes

The following is a detailed description of the events requiring logging in the course of a transactional process, as depicted in Figure 9. In essence, our logging scheme follows that of conventional 2PC with minor modifications that are cited whenever necessary. At each step, the effects of failures (denoted by f_i) are considered.

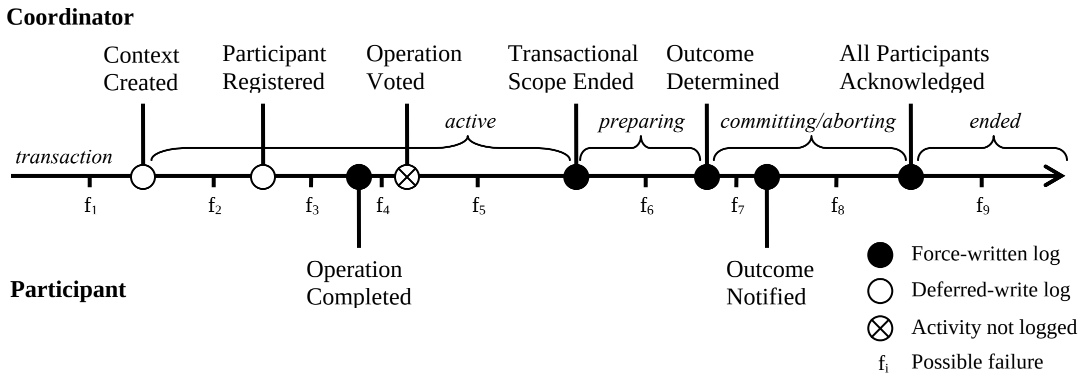


Figure 9: Events logged during a Web service transaction

5.2.1. Context Created (Coordinator)

The coordinator responds to the request to initiate a new transaction by creating *a coordination context* which gets assigned a unique transaction identifier. A log record must be created to mark the creation of a new transaction. However, force-writing is unnecessary as no participants can possibly be registered yet and therefore, no real work has been done. If the coordinator crashes before the record reaches stable storage (f_1 in Figure 9), no evidence of the transaction will exist, but no work is lost either. The initiating process will have to retry the transaction once the coordinator becomes available. On the other hand, if the coordinator crashes after the record is persisted (f_2 in Figure 9), recovery will only restore the new coordination context before resuming normal operations.

5.2.2. Participant Registered (Coordinator)

Prior to performing any useful work, participants must register with the coordinator by supplying their endpoint address. This address is stored in a log record at the coordinator. If log force-writing is avoided, the worst possible scenario is that in which the coordinator crashes after some participant performs an action, but before its corresponding endpoint record reaches stable storage (f_4 in Figure 9), causing it to "forget" about pending work. Fortunately, in such an early stage, participants can unilaterally decide to abort their work by means of a timeout. If the coordinator crashes before participants execute any operations (as in f_3 in Figure 9), participants will eventually fail to report their outcome and make the same unilateral decision. Therefore, since the transaction can still be aborted by all participants, log force-writing can be avoided.

5.2.3. Operation Completed (Participant)

Participants fulfill parts of the transactional process and report their outcome to the coordinator. Before voting, they must ensure that the operation's outcome is present on stable storage. For atomic transactions (WS-AT), if the operation yielded a positive outcome, database activity log records must also be forced; otherwise, participants could not guarantee the recoverability of such actions. In contrast, for operations voted negatively or read-only, logging of database activity can be omitted as these do not have to be re-executed upon recovery. After this event is logged (f_4 in Figure 9), participants can recover their state plus all operations voted positively. If the coordinator fails past this point, participants will be able to resubmit their votes as part of its recovery process, regardless of any site failures.

5.2.4. Operation Voted (Coordinator)

Whenever a participant votes (using either standard 2PC or unsolicited votes), the coordinator has no need to write any log records: after a failure (f_5 in Figure 9), it can restore its transaction state representation by polling all registered participants. This saves logging overhead and avoids keeping duplicate information, at the cost of additional network messages in case of failure.

5.2.5. Transactional Scope Ended (Coordinator)

Once the initiating process notifies the coordinator about the end of the transactional scope, the transaction protocol begins. A forced log record marks the change of transaction state from *active* to *preparing*, allowing a recovering coordinator to determine whether to wait for additional operations to be invoked by the process (f_5 in Figure 9), or to re-execute the transaction protocol after the interruption caused by the failure (f_6 in Figure 9). Note that although log force-writing could be avoided (as participants can still abort on their own), the savings are outweighed by the possibility of loosing the work of an entire transactional scope.

5.2.6. Outcome Determined (Coordinator)

The transaction's outcome is decided once all votes are evaluated. Whichever the outcome, the change on transaction status must be immediately reflected in the log by switching it from *preparing* to *committing* or *aborting*. In this way, the effects of the transactional process are guaranteed to be consistent with the recorded outcome. If the coordinator fails before the updated record reaches the log, its recovery is as described in the previous event. If the failure occurs beyond this point (f_7 in Figure 9), the coordinator checks its log to find the original decision and resumes the protocol accordingly; any participants in which the protocol had already finished receive the same final instruction and simply acknowledge it.

5.2.7. Outcome Notified (Participant)

The coordinator directs each participant to complete their part of the work by committing or aborting operations. Participants must immediately log the final decision and carry out the necessary actions to honor it. If the decision is to commit, there is a narrow yet possible scenario (somewhere near f_8 in Figure 9) in which the *commit* record is logged, but due to a precisely-timed failure, the corresponding *commit* command is not executed on the local database. Upon recovery, the participant will act as though the commit had actually occurred, causing global transaction inconsistency. To detect this condition, a second log record (*committed*) is created following the database commit; this way, if a recovering participant finds any operations in which the *commit* record exists but their *committed* counterpart is missing, the user is notified, as the subtransaction might have not been committed. Note that if the decision is to abort, there is no risk of reaching a similar scenario: If the *abort* record reaches the log, but the corresponding database command is not executed, a site failure will anyway cause the pending operation to be implicitly undone by the local database, and thanks to the log record, participant recovery will not re-execute the operation. Finally, if the initial commit or abort log record does not reach the log, participant recovery is as described in Section 5.2.3, followed by a second outcome notification from the coordinator (if unsolicited votes are used), or the participant's request to *replay* the last message. Participant recovery is analyzed in detail in Section 6.

5.2.8. All Participants Acknowledged (Coordinator)

Once all participants have acknowledged the command to finalize the transaction, the coordinator forgets its state. As in the previous event, the transaction's log record must be immediately updated by changing its status to *ended* (f_9 in Figure 9). In this way, the transaction is no longer considered for recovery.

6. System Recovery

Our approach to recovery attempts to let a running process resume operation following a temporary failure of one or more of its members. Just as with logging, our recovery procedures are largely based on the algorithms for 2PC protocol recovery, and influenced by existing work, particularly, recovery using *Logical Logging* [15], the *Generic Log Service* [40] and the *Durable Scripts Containing Database Transactions* [29]. During recovery, transaction participants are forward-recovered to the point of failure; then, by resuming normal execution, the transaction is likely to reach a correct state.

6.1. Coordinator Recovery

The coordinator's recovery procedure is triggered following a failure, before this service becomes available to new transactions. During the process, the coordinator may in turn direct participants to recover, regaining control of the transaction even in cases of multiple site failures. It emcompasses the following steps:

Application State Recovery

1. Enter into a temporary state called *recovering*: All external requests are ignored while the coordinator recovers. Entities interacting with the coordinator must understand this state and retry their calls later.
2. Read from the *transactions* log relation (Figure 7) looking for all transactions whose status is other than *ended* or *in-doubt* (refer to Table 3 for a comprehensive list of transaction states). In-doubt transactions are not recovered because they are considered to require user intervention to be completed.
3. For each unfinished transaction, create a new coordination context using the unique transaction identifier (*transactionID*), coordination type (*coordinationType*) and participant endpoints (*endpointURL*) recorded in the log.

Once this stage is completed, the coordinator has restored the coordination context for each transaction affected by the failure, as well as the necessary information to recover the state of their transaction protocol.

Transaction Protocol Recovery

4. Depending on the status at which the transaction was left:
   - If it was *active* or *preparing*:
     a. Ask every registered participant to send its vote once again by invoking their *prepare* operation. This must be done because participant votes are maintained only in main memory (see Section 5.2 for details on transaction event logging optimizations).
        - If the participant is available, it queries its own log and reports the requested information.
        - If the participant cannot be contacted, retry a predefined number of times. Assuming that it becomes available soon enough, and if coordinator-initiated recovery is active (explained in Section 6.2), ask the participant to *recover*. Following their recovery, participants query their local log and report the requested information.
        - Otherwise, if the participant remains unreachable, decide to abort the transaction and log this decision.

     b. If the transaction was *active*, the business process may not have finished invoking all service operations. Therefore, wait for it to signal the completion of the transactional scope.

     c. Otherwise, if the transaction was *preparing*, carry out the transaction protocol: Make a decision based on gathered votes, log it, and confirm or cancel operations at every participant.
   - If the transaction failed past the decision point (status value of *committing* or *aborting*):
     a. Read the final decision from the log.

     b. Finish the transaction protocol in all participants that may need to do so. Retry to contact unreachable participants and ask them to recover beforehand (if configured to do so). If any participant remains unreachable, mark the transaction as unmanageable (*in-doubt*) and notify the user.

        Since the coordinator had already started the final phase of the protocol before the failure, it is likely to find a combination of finished and unfinished participants. Whichever the case, the coordinator can safely request all participants to finalize their work: If they have already done so, they simply acknowledge the request. If they have not, they confirm or cancel their work as they would under normal conditions.

5. For every transaction left unfinished, schedule a timeout job that will abort it in case it is not finished within the expected time frame. Such time period can (and, in general, should) be configured separately for different coordination types.

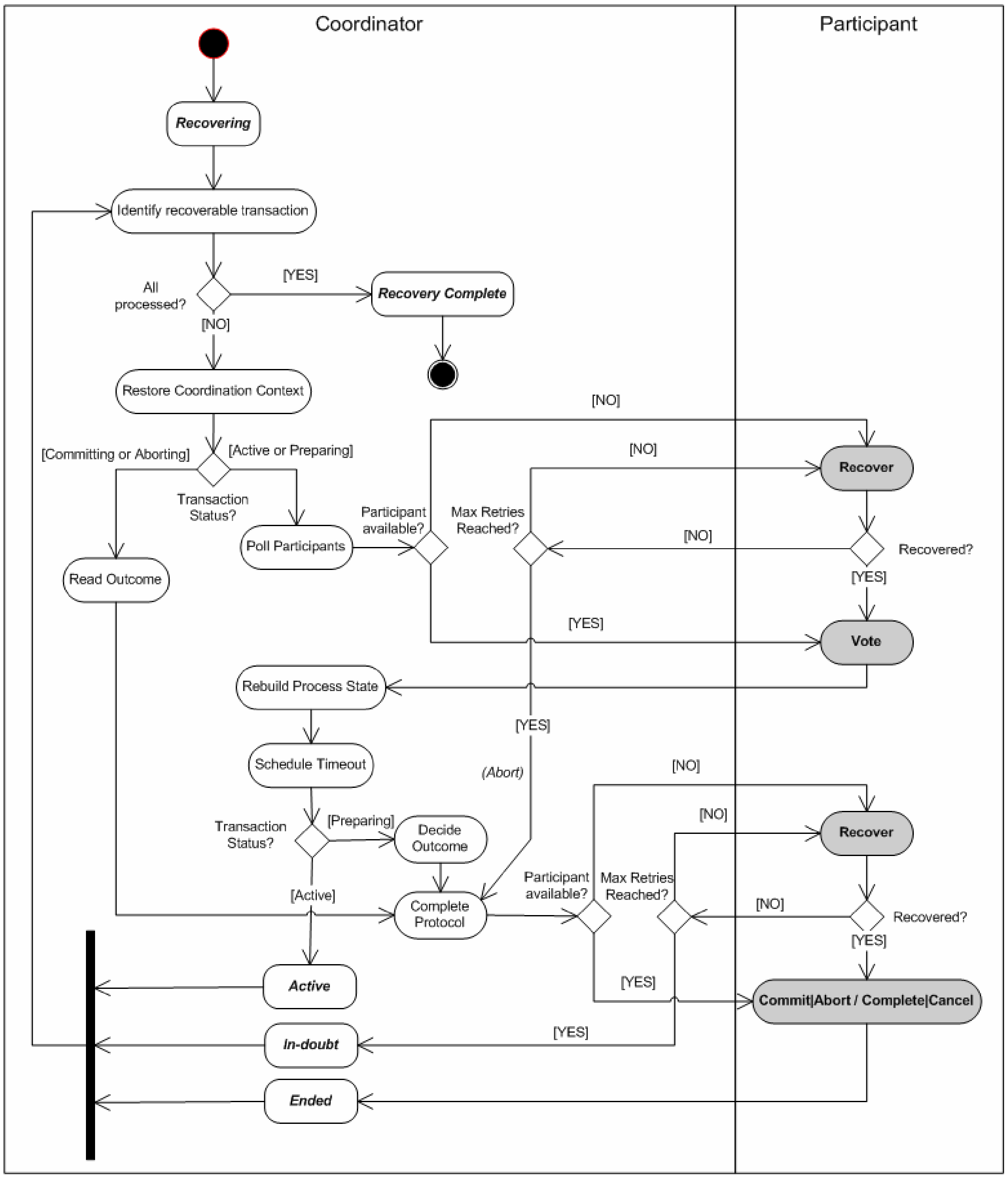


Figure 10: Activity diagram for Coordinator recovery

The activity diagram of Figure 10 summarizes the coordinator recovery procedure assuming that coordinator-initiated recovery (described in the next section) is used.

It is worth noting that from the coordinator's perspective, a participant has failed when an exceptional condition is found during its invocation. In general, these may be caused by network partitioning or site failures (see [8] for a complete classification of failures). However, in the former case, participants do not require recovery as their internal state is not altered. Clearly, participants are responsible for evaluating the actions to be taken upon receipt of a request for recovery.

By capturing the original message that triggered the transactional business process, the coordinator is capable of restarting it. This constitutes a last alternative to recover the system from an orchestration engine that fails in the middle of the transaction, particularly, before the transaction is decided. In such a case, the coordinator aborts the incomplete transaction and attempts to restart the business process as though it were the process' client application, leading indirectly to the creation of a completely new (yet possibly equivalent) transaction.

If the coordinator fails for a period of time exceeding the transaction timeout, a participant that has not received the coordinator's final decision (due to either a lost message or the participant's failure) may be indefinitely blocked. Therefore upon a timeout, such a participant will need to determine the coordinator's decision first by asking another participant. If no participant knows the decision, a new coordinator may be elected [25]. Note that these two options are left as future work for our framework's prototype.

6.2. Participant Recovery

Traditional recovery algorithms for distributed transactions (e.g., 2PC recovery [3]) suggest that participant recovery is self-initiated, occurring as soon as the hosting site becomes available. Nevertheless, in an active Web of independent services, where there is a significant likelihood of communication and site failures, it is possible to advocate for a *just-in-time* approach to participant recovery. The reason is that participants lack a global perception of transaction state, so their recovery is not useful unless the coordinator is available and ready to resume transaction processing. Also, considering that participants may take part in transactions managed by different coordinators, it is not optimal to recover all transactions at once, as some of them may have to wait for their coordinator. This approach may be particularly beneficial to WS-AT transactions, where recovery involves acquiring locks on underlying data, reducing the concurrency of the involved sites. Since both the traditional and the proposed approaches present their advantages and disadvantages depending on domain-specific factors (e.g., required coordination type, degree of local data contention, average transaction lifespan, probability of failures, etc.), our framework makes available the following options:

- Participant-initiated recovery: Immediately after a crash, participants restore the state of all pending operations for all transactions. This is the traditional approach.
- Coordinator-initiated recovery: Participant recovery is exclusively triggered by the coordinator and limited to specific transactions. Following a crash, participants take no particular action.

Given that both alternatives yield the same final effect (i.e., recovery of failed transactions), differing only on the time when recovery occurs, they are equally effective in providing fault-tolerance. For brevity, we describe the participant recovery procedure using coordinator-initiated recovery; normal recovery is in essence very similar and widely documented in existing literature (e.g., [3] and [25]).

Application State Recovery

1. Begin transaction recovery when a message requesting the *recover* operation is received. The message must contain the transaction's identifier, coordination type and coordinator endpoint URL. Enter into the *recovering* state so that all external requests targeting the transaction are momentarily ignored.

2. Validate the request for recovery by determining whether a coordination context exists for the requested transaction. If it does, the request may have been triggered only as a result of network partitioning and therefore no recovery is needed.
3. Restore the local coordination context. Populate it with the transaction's unique identifier, coordination type and coordinator endpoint URL.

Transaction Protocol Recovery

4. Retrieve from the *operations* log relation all activities performed by this participant on behalf of the failed transaction.
5. Depending on the presence or absence of a finalization record (*committed* or *aborted* in WS-AT; *closed, canceled* or *compensated* in WS-BA):
   a. If such a record exists, the participant was able to finish its part of the transaction before the failure. The state of its local database accurately reflects the decision taken by the coordinator and thus no further actions are required.
   b. If the coordination type is WS-AT and the finalization record is not present, but an attempt to commit was made (indicated by the presence of a *commit* record), then the operation might have not been actually committed. Since the only way to determine this is by examining the underlying data, this condition terminates recovery with an exception, causing the coordinator to switch the transaction's status to *in-doubt* and to notify the user about the anomaly.
   c. Otherwise, there are pending operations at the participant that must be recovered. Depending on the coordination type:
      - For atomic transactions (WS-AT), verify whether uncommitted database activity must be replayed by examining all operations whose *outcome* was positive (i.e., vote to commit). If so, reconstruct the original database operation as follows:
        a. Retrieve the original *command* from the *operations* relation, and prepare it for execution using a suitable programming object. Choosing such an object depends on the type of database command (e.g., SQL statement, function call) and data access implementation.
        b. Augment the above object to include associated parameters. Set their properties based on the information in the *parameters* log relation (using its *type*, *direction* and *value* attributes).
        c. Execute the call without committing (this will be done once the final decision is received).
      - For long-running transactions (WS-BA) no immediate action is required. All tentative operations were originally made permanent in the local database.
6. If pending operations were found, wait for the coordinator's final decision. No other action is possible because the participant is in its *uncertainty period* [3]: It does not know whether the coordinator will decide to commit or abort (no communication among participants is assumed). Hence, from this point on, protocol interaction and event logging resumes normally once the business process resumes execution. The activity diagram of Figure 11 summarizes the participant recovery procedure.

To accurately reproduce database activity lost during a site failure, the log is used to record and replay operations in original execution order. In line with previous work [32], we assume that replaying uncommitted database activity lost after a failure is likely to yield the original results. However, since the state of the underlying data may change between the time of failure and the time of re-execution, it is not always possible to guarantee identical results. A domain-specific assessment must be done prior to adopting this recovery approach for a given application.

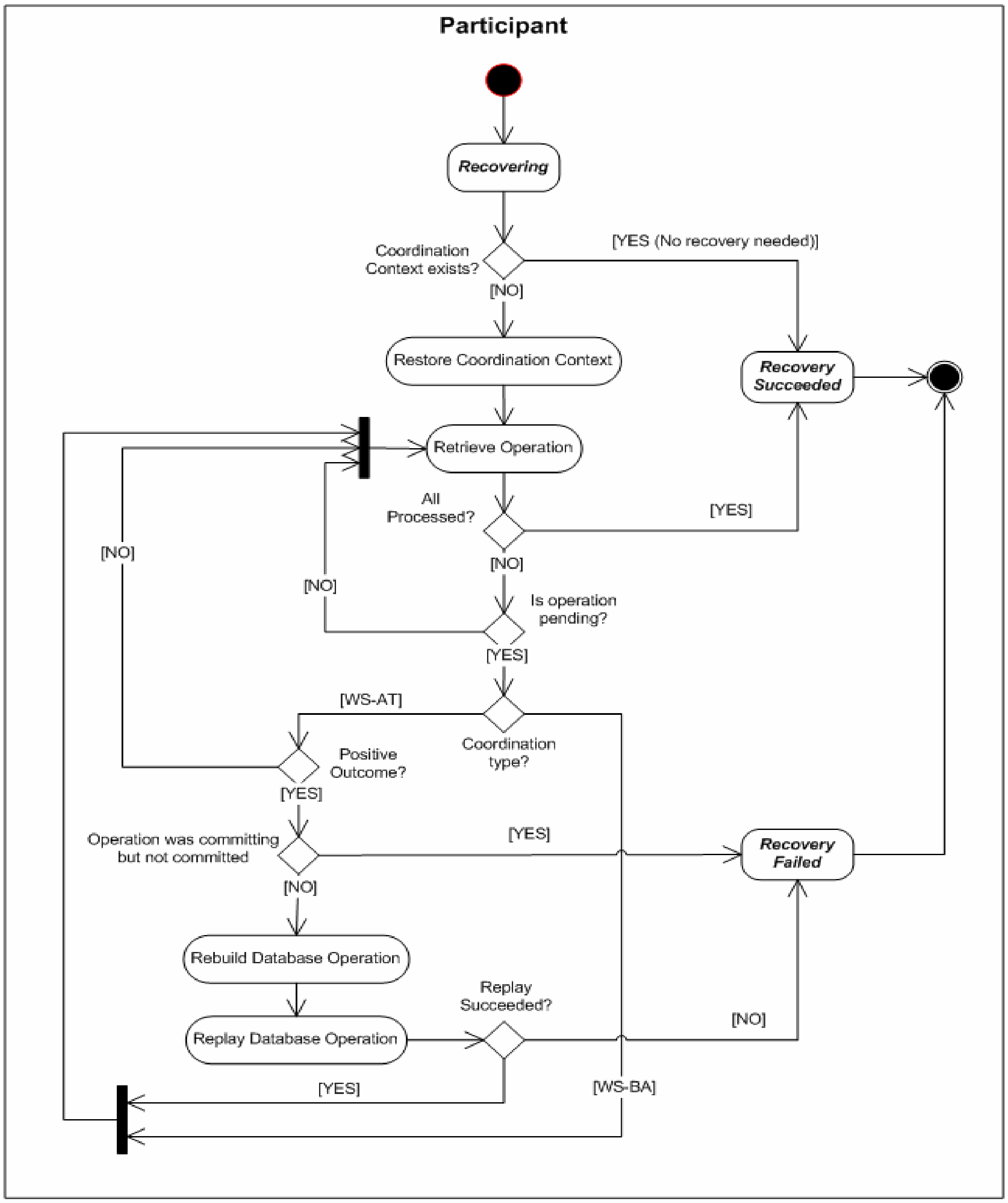


Figure 11: Activity diagram for Participant recovery

Under certain conditions, the recovery of atomic transactions may lead to application blocking or deadlocking. If two or more transactions affecting the same data are simultaneously recovered, the nature of ACID transactions will inevitably lead to blocking until the lock-holding transaction commits or aborts its operations. Fortunately, since recovered transactions are also subject to expiration, they are either completed or eventually aborted, giving other transactions the opportunity to work with the contended data.

In order to take advantage of transactional fault tolerance, it is of vital importance that business process definitions use fault handling and retry logic, allowing transactional services to recover and resume normal operation, as opposed to breaking the regular flow of process events. In BPEL, this can be done using process scopes and fault handling constructs [18].

6.3. Log Checkpointing

As with other logging mechanisms, in the long term, recovery time can be reduced by means of periodic log checkpoints. In order to evaluate the effects of long-term log growth, we decided to use *transaction-consistent checkpointing* [9] on coordinators as well as participants. In our prototype, checkpointing is done as follows:

- A configuration property that we call *checkpoint trigger point* controls the number of transaction log records at which the checkpoint will be attempted. The user should decide an appropriate value depending on the expected workload (i.e., transactions per unit of time).
- At runtime, a task periodically forces all pending log writes to stable storage and checks the number of log records (i.e., *transactions* for coordinators, *operations* for participants). Whenever the number exceeds the checkpoint trigger point, the service will stop accepting new transactions and wait for existing transactions to finish.
- Once the transaction-consistent point is reached, the contents of the logging schema are flushed. At coordinators, this is a safe action because, in order for a transaction to be switched to *ended*, all participants must have acknowledged the final decision; since all transactions must be in this state, no log records are needed for recovery. At participants, this action is also safe because a finalization record (*committed* or *aborted* in WS-AT; *closed, canceled* or *compensated* in WS-BA) can only be found once both the coordinator and the participant have carried out all the necessary actions for a given transaction.

It must be noted that in the presence of long-running or in-doubt transactions, this algorithm will struggle to find an appropriate time window to flush the log. Future releases of the framework may implement a more effective algorithm (e.g., based on fuzzy checkpoints).

7. Related Work

Transactional Web services are a multidisciplinary subject involving areas such as Web service composition, workflow systems, distributed transactions and fault tolerance, all of which constitute the foundation of our work. In this section, we present a discussion on how this work differs or extends some of the previous work in these areas.

7.1. Web Service Composition

Recent work in [34] suggests how specifications of the Web services stack, in particular BPEL, WS-C and the associated transaction specifications, can be combined to provide coordinated Web processes. Our work adopts their declarative definition of coordination scopes as part of the process specification, as opposed to using programming interfaces for a specific language (e.g., the Java Transaction API). Meanwhile, [13] delves into the technical implications of adapting existing transactional applications into the Web services model; however, no working prototype was presented at the time. Recently, [31] advocates extending BPEL to provide distributed coordination features, making the WS-BA specification redundant and therefore, unnecessary. We follow a more standards-based approach by focusing our effort on implementing the WS-AT and WS-BA specifications without suggesting modifications to the BPEL specification.

Probably the most related approaches to recovery come from [36] and [20], although their work differ from ours in subtle yet important ways: [30] tackles the problem of service reliability using forward recovery, but their *composition actions* (predefined actions with a well defined behavior, both in absence and presence of failures) are statically defined for every participant by specifying, in a special XML language, the actions that must be carried out in case of failure. Instead, our approach to failure recovery attempts to automate these actions whenever possible (e.g., when re-execution of submitted tasks is required), limiting user intervention to cases when domain-specific actions must be taken (e.g., when compensation is needed). [20] proposes a framework of *action ports* (proxy Web services) that negotiate and enforce transactional behavior in existing services. Despite suggesting the need for logging critical interactions, it does not explicitly mention any particular mechanisms for fault tolerance, which is a key contribution of our work.

Previous to the advent of the Web services specifications, many distributed systems relied on implementations of the CORBA Object Transaction Service and DCOM specifications (e.g., systems based on the Java Transaction Service or Microsoft's Distributed Transaction Coordinator). Unlike Web services, which use XML-formatted messages for communication, these systems rely on specific protocols (e.g., Internet Interoperable ORB Protocol) to exchange data in suitable ways according to the programming language (e.g., stubs and skeletons, remote procedure calls, etc.). Several database systems have made available an XA-compliant interface [42], which provides common interface that allows external transaction coordinators to direct their participation in distributed transactions.

Industry implementations of the Web Services Transactions specifications include IBM's Web Services Atomic Transaction for WebSphere[5], JBoss Transactions[6] (one of the earliest implementations, originally developed by Arjuna), Microsoft Indigo[7] (which aims to integrate Web service specifications with an upcoming operating system), and Apache Kandula[8] (an open source project dedicated to providing capabilities integrated with existing transaction managers). To date, most of these implementations are still undergoing acceptance and interoperability tests, having limited amounts of publicly-available documentation. Nevertheless, a common assumption appears to be the ability or willingness to make code-intensive modifications on existing applications, which may not necessarily hold in all cases. In contrast, our work intends to deliver reliable transactional features while requiring as few changes to the application as possible (see Section 9 for an evaluation of required changes).

7.2. Workflow Systems

A comprehensive discussion on transactional workflows can be found in [27]. There, the fundamental differences between *multidatabases* and *distributed databases* are explained. The former kind characterizes the types of systems which are the subject of our discussion: independent processing entities containing related data, yet whose underlying resource managers do not provide support for distributed activities. The authors also introduce the concept of *failure atomicity*, or application-dependent requirements for the workflow to be left in an acceptable termination state. Consequently, the goal of failure recovery is to enforce a workflow's failure atomicity by restoring its context from information kept on stable storage. Assuming that a system is able to perform forward recovery and that participant operations yield identical outcomes, process execution may continue by resubmitting affected tasks

[5] http://www.alphaworks.ibm.com/tech/wsat

[6] http://www.arjuna.com/products/arjunats/

[7] http://msdn.microsoft.com/webservices/indigo/

[8] http://ws.apache.org/kandula/

automatically. Such assumptions are also used in our framework in order to deliver recovery of database activity for atomic transactions.

Later, in [21] the authors address error handling and recovery in WebWork, an early web-based workflow engine. Their study classifies task types (i.e., human-computer, Web, transactional and non-transactional) in terms of the difficulty they present for recovery. Ten types of errors were identified, ranging from data entry errors to client and server failures. While our work shares identical goals, it is focused on protecting against operational failures. Errors in the business logic are expected to be addressed by the client applications, the policies and validations enforced by individual services, and the business process definition.

In the area of Web service composition, [35] introduces the concept of *consistency units* (C-unit), useful when workflow step instances cannot be determined in advance. C-units are transaction-style sections of a process that enforce data integrity constraints across participant boundaries. Although C-units can be locally committed as soon as they finish execution, resources must still be retained until the workflow terminates (in order to preserve data consistency), even at the cost of decreased concurrency. On the other hand, transactional scopes demarcated in BPEL may either commit local transactions at the end of the process (WS-AT) or commit immediately without blocking (WS-BA) as it is necessary for long-running transactional processes.

7.3. Distributed Transaction Models

A comprehensive study of Web service and transactions is given in [26] as part of proposing the *business transaction framework* (BTF), which characterizes the needs of business processes conducted over loosely coupled environments. It recognizes the need for new, unconventional kinds of atomicity criteria and the importance of auditing, logging and recovering transactions.

Besides Web Services Transactions specifications, other workflow-centric transaction specifications have been proposed:

- The Business Transaction Protocol (BTP) [23] suggests the need for two types of interactions, namely *atoms* or short transactions with full ACID properties, and *cohesions* or longer, non-atomic transactions built by aggregating the former type. BTP assumes that failures will occur, so it allows participants to make autonomous decisions under special conditions. It cites the importance of distinguishing between communication failures and node failures, helpful in determining whether participant state information must be restored.
- The Web Services Composite Application Framework [24] has close similarities with WS-C, WS-AT and WS-BA; however, it further divides the problem into context (WS-CTX), coordination (WS-CF) and transaction management (WS-TXM). Regarding fault tolerance, recovery is a joint responsibility of the framework and the application, and states that in large, complex transactional environments recovery may not always be automated, so manual intervention may be necessary to restore application consistency.

Extended transaction models like those proposed by WS-BA and the aforementioned efforts have been influenced by earlier transaction models, namely Sagas [5], Flexible Transactions [28] and ConTracts [39]. Sagas were conceived in response to the concurrency problems raised by long-running transactions. For each business operation, sagas prescribe the creation of a corresponding *compensating subtransaction* that cancels its effects. Likewise, Flexible Transactions relax isolation and atomicity requirements, but unlike Sagas, they make it possible to reach a positive outcome even when some of their subtransactions may abort, just as proposed by the *MixedOutcome* coordination type [17]. A key aspect of ConTracts, on the other hand, is its recoverability: The state of a series of actions (known as *script*) can be recovered for every action that was active when the failure occurred, allowing the Contract to continue its execution (i.e., forward recovery).

7.4. Fault Tolerance and Reliability

Distributed application recovery and Web service reliability are topics that have been extensively covered in the past. Even though the techniques used by the proposed logging scheme are the results of existing research, to the best of our knowledge, their use has not been applied in this context.

In [14], the authors developed the concept of an *installation graph* which explains the order of log writes necessary to forward-recover the state of a database at the time of a crash. Subsequently, [15] introduced the notion of *logical log* operations, which not only reduce the volume of records that have to be written, but also allow for more generic logging schemas. In the context of databases, [29] explains how *Durable Scripts Containing Database Transactions* (DSDT) can be used to record short ACID transactions that may have to be replayed later as a means to restore an application's state. Our experimentation (presented in Section 9) shows that the combined use of these contributions constitutes an effective way to recover database activity on WS-AT transaction participants.

Transparent logging is a desirable feature when it is meant to be incorporated into existing applications, as it minimizes the number of required application changes. Towards that goal, the *Generic Log Service* described in [40] can accommodate the logging of protocols represented by finite state machines, and in particular, transactional protocols such as 2PC. Earlier research [2] stated the need for fault-tolerance in distributed applications beyond the guarantees provided by underlying resource managers. In order to avoid the need for fault-tolerance logic in every application, the authors analyzed the feasibility of transparent recovery solutions, allowing one to transform a non-resilient application into a resilient one. Our prototype framework intends to provide such a feature by making available a programming interface which can be used to easily transform existing services.

8. Prototype Implementation

Our prototype framework was implemented in Java and relies entirely on open source projects: Web services run on Apache Axis[9] and Tomcat[10]. The logging facility uses an embedded database system, BerkeleyDB[11], which minimizes resource utilization and simplifies deployment. Sample processes were modeled using METEOR-S[12] BPEL Process Designer and run in ActiveBPEL[13], an execution engine that supports process recovery. The sample Web services access data from any source for which a JDBC driver is available; example database schemas for PostgreSQL, MySQL, Oracle and SQL Server are available as part the prototype distribution. Log4J's Chainsaw[14] and ActiveBPEL's graphical process view are recommended tools for monitoring transaction progress.

To minimize the impact on application code, we leverage Axis' architecture of handlers and handler chains, which help to automate certain protocol-specific tasks. A *handler chain* is a pre- or post-

[9] Apache Axis, http://ws.apache.org/axis

[10] Apache Tomcat, http://jakarta.apache.org/tomcat

[11] BerkeleyDB, http://www.sleepycat.com/products/je.shtml

[12] METEOR-S, http://lsdis.cs.uga.edu/projects/meteor-s

[13] ActiveBPEL, http://www.activebpel.org

[14] Apache Log4J, http://logging.apache.org/log4j

processing pipeline executed on every Web service request, in which individual tasks are performed by a particular *handler*. In the prototype, a pre-processing handler chain is responsible for message correlation (described next) and participant registration, while a post-processing chain is responsible for reporting operation outcomes to the coordinator (i.e., voting) whenever appropriate. This way, services are insulated from the additional procedures required by the coordination protocols every time an operation is performed.

Message correlation is the task by which incoming messages are routed to the appropriate service conversation, enabling participants to carry out multiple interactions during the lifetime of a given process. Correlation relies on a unique conversation identifier present on every protocol message (e.g., a *transactionId*), which is matched to an equally-identified coordination context in the target service, if one exists. In BPEL, this task requires using *correlation sets* [18]: language constructs defining the message attributes that the runtime engine must evaluate whenever a message is received, so that it can be routed to the appropriate process instance. Asynchronous calls, which are especially important for long-running business activities, are possible by combining message correlation and one-way service invocations.

To optimize logging performance, the prototype's embedded database uses a single set of files on every site where the framework is deployed. The logging facility administers concurrent access to the log for all participants and coordinators hosted on a given site. Log record writing is in itself atomic: Records are either fully written or not written at all, as the logging database performs file recovery in the event of a failure.

The framework's behavior can be changed to suit particular application requirements by means of configuration. Several configuration options control aspects such as transaction timeouts for a given coordination type, the number of failed attempts before requesting participant recovery, the physical location of the logging device and the utilization of logging optimizations covered in Section 5.2.

9. Empirical Evaluation

The objective of this evaluation was to validate the features of our prototype framework implementation with regards to its effectiveness, performance under normal and failure conditions, and number of required application changes. Experimentation was done with variations of the well-known travel agency use case, involving flight reservation, hotel reservation and banking Web services. These services contained business logic accessing data in either PostgreSQL or MySQL databases.

The services were used to compose three different versions of a BPEL process: A non-coordinated process (depicted in Figure 12), a coordinated process in which activities were enclosed in a WS-AT transactional scope (Figure 13), and a similar process in which a WS-BA scope was used. For the last two, the coordinator service (described in Section 4.1) was made available and the intervening services were enhanced using the *participant* framework entity (described in Section 4.2).

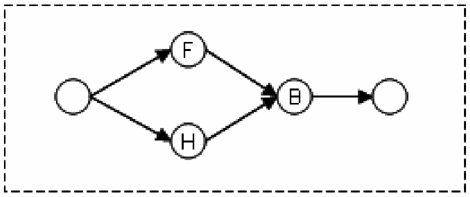


Figure 12: Non-coordinated process

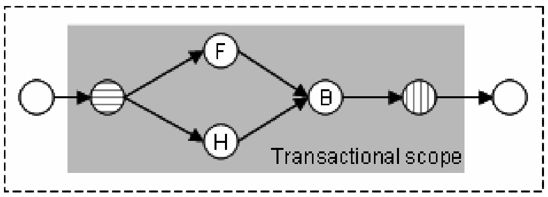


Figure 13: Coordinated process

To simulate a possible deployment scenario, we used three Pentium 4-based computers connected over a local area network. The first computer hosted the client Web application from which the business process was initiated. A second computer was used for process orchestration. The third computer hosted all participant services as well as the coordinator.

To establish an initial performance comparison, we measured the execution time for each process from the perspective of the client application (i.e., the time elapsed between process invocation and reception of the return call). Execution times were averaged after 500 repetitions, obtaining 287, 1045 and 1010 milliseconds (ms) for the non-coordinated, WS-AT and WS-BA processes, respectively. Then, the coordinated processes were run once again with logging optimizations enabled, obtaining new averages of 1019 ms for WS-AT and 1003 ms for WS-BA (Figure 14). These marginal improvements are due to the fact that these optimizations only affect a couple of transaction events (outlined in Table 5). Nonetheless, for WS-AT this time difference may become desirable considering its collective effect on a highly active system.

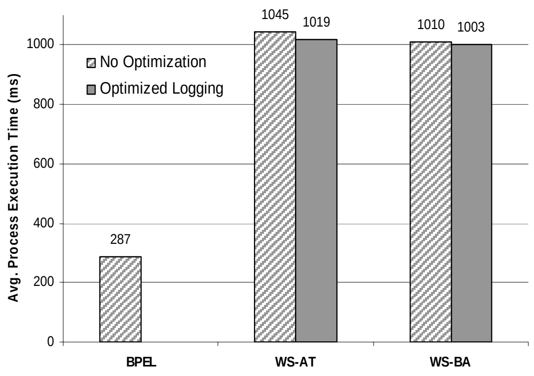


Figure 14: Total execution times

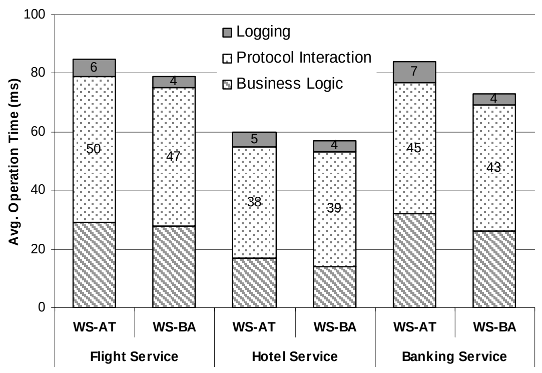


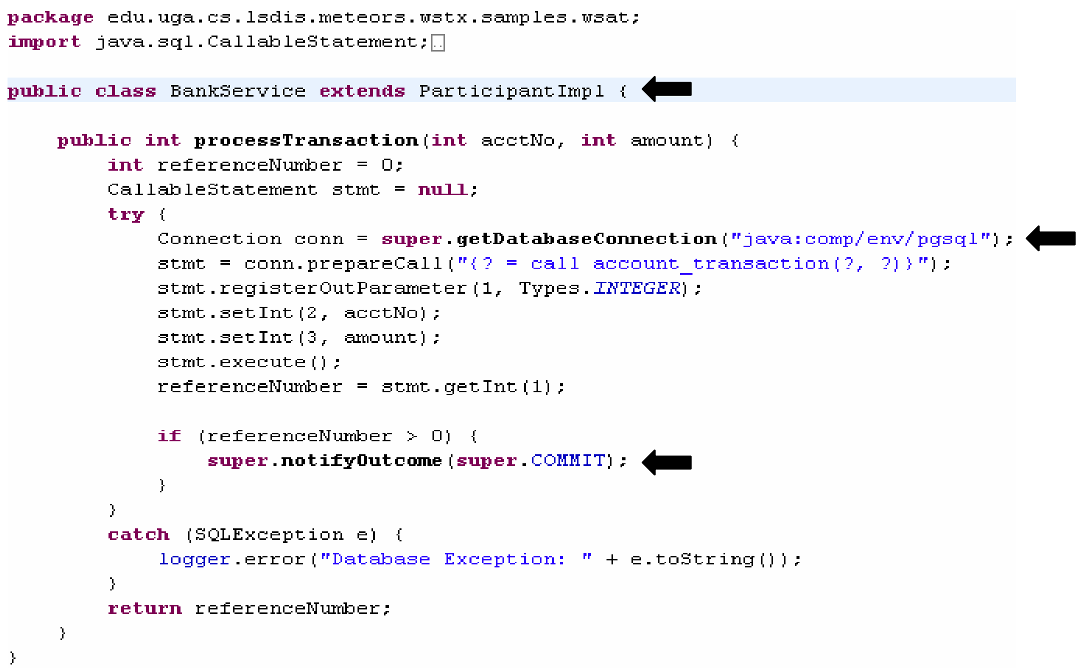
Figure 15: Overhead introduced by logging

On average, transaction event logging resulted in an overhead of 5ms for WS-AT and 4 ms for WS-BA (this difference is due to the absence of database activity logging in the latter). Logging, on average, made up 7.4% of the operation completion times, as shown in Figure 15. Meanwhile, protocol-related interactions (i.e., registration and voting) performed as part of these operations represented 61% of their completion time. This high percentage can be understood considering that protocol interactions involve communication with the coordinator, and in turn, event logging (e.g., during participant registration). It must be noted that this behavior is dictated by the reference specifications, rather from a particular design or implementation decision.

To establish the prototype's fault tolerance effectiveness, we simulated random failures at any stage of the transactional process. We were particularly interested in observing the behavior explained in Section 5.2.7, in which certain failures cannot be managed by the framework and thus, require user intervention. After analyzing the results from 500 repetitions, we found that only 1.6% of the tests led to this specific condition. Meanwhile, 42.2% of the test cases caused the processes to be backward recovered by the transaction protocol. In the remaining 56.2%, the processes were recovered to the point of failure and continued normally thanks to fault tolerance. Clearly, under identical conditions, implementations lacking these reliability features would result in a total 57.8% of the cases left in an inconsistent state.

A common factor affecting coordinator recovery and participant-initiated recovery (explained in Section 6), is the time required to examine the log while searching for unfinished transactions. In our testing, we found that it takes an average of 308 ms to examine every 500 log records, which helps to establish an approximate lower bound for recovery time. Since this is a slow operation, log checkpointing is critical to ensure the continued efficiency of recovery.

We managed to reduce the number of required application code changes down to two or three instructions, depending on the coordination type: One change is required for inheriting from the *participant* framework class. Another change must be introduced for communicating the operation's outcome to the framework. For WS-AT, an additional line enables the application to transparently log database activity. Figure 16 shows a sample Web service after conversion.



```
package edu.uga.cs.lsdis.meteors.wstx.samples.wsat;
import java.sql.CallableStatement;

public class BankService extends ParticipantImpl {

    public int processTransaction(int acctNo, int amount) {
        int referenceNumber = 0;
        CallableStatement stmt = null;
        try {
            Connection conn = super.getDatabaseConnection("java:comp/env/pgsql");
            stmt = conn.prepareCall("{? = call account_transaction(?, ?)}");
            stmt.registerOutParameter(1, Types.INTEGER);
            stmt.setInt(2, acctNo);
            stmt.setInt(3, amount);
            stmt.execute();
            referenceNumber = stmt.getInt(1);

            if (referenceNumber > 0) {
                super.notifyOutcome(super.COMMIT);
            }
        }
        catch (SQLException e) {
            logger.error("Database Exception: " + e.toString());
        }
        return referenceNumber;
    }
}
```

Figure 16: Sample Web service after conversion (arrows denote required changes)

10. Conclusions and Future Work

The fundamental properties of transactional business processes cannot be guaranteed unless the intervening participants offer minimum guarantees about their reliability. Our work analyzed the fundamental fault tolerance requirements for Web services in such a transactional environment. Based on increasingly popular specifications, we proposed a framework that leverages existing logging and recovery techniques to enable failure recovery. In this way, services become capable of restoring critical state information and pending database activity, allowing business processes to resume normal operation past the point of failure.

Unlike previous research, which proposes the use of proxy services or code-intensive modifications, a prototype implementation showed that the proposed framework can be easily integrated into existing services. Our experimentation found that even though transaction coordination results in a significant increase in the complexity of service interactions, the overhead associated with supporting fault tolerance is comparatively minimal. Most importantly, we established that our logging and recovery schemes are effective in all but a limited number of failure scenarios previously analyzed.

In the immediate future, we want to investigate the additional reliability requirements that transactions resulting from more complex business processes impose on our framework. Such transactions are likely to involve advanced transaction models, whose implications might not yet be explicitly stated in the existing specifications.

Transactional business processes could also improve their resilience by approaching failures using autonomic recovery. By enabling transaction coordinators to discover alternative participants, this option would greatly expand the options currently available to the framework, especially under permanent failure scenarios. Recent METEOR-S work on Autonomic Web Processes [38] can be applied to address this and various other problems.

CHAPTER 3

CONCLUSIONS AND FUTURE WORK

The fundamental properties of transactional business processes cannot be guaranteed unless the intervening participants offer minimum guarantees about their reliability. Our work analyzed the fundamental fault tolerance requirements for Web services in such a transactional environment. Based on increasingly popular specifications, we proposed a framework that leverages existing logging and recovery techniques to enable failure recovery. In this way, services become capable of restoring critical state information and pending database activity, allowing business processes to resume normal operation past the point of failure.

Unlike previous research, which proposes the use of proxy services or code-intensive modifications, a prototype implementation showed that the proposed framework can be easily integrated into existing services. Our experimentation found that even though transaction coordination results in a significant increase in the complexity of service interactions, the overhead associated with supporting fault tolerance is comparatively minimal. We also established that the time required to recover a transactional process varies depending on the stage at which failures occur, the kind of transaction (i.e., atomic or long-running) and the efficiency of log management. Most importantly, we established that our logging and recovery schemes are effective in all but a limited number of failure scenarios previously analyzed.

In the immediate future, we want to investigate the additional reliability requirements that transactions resulting from more complex business processes impose on our framework. Such transactions are likely to involve advanced transaction models, whose implications might not yet be explicitly stated in the existing specifications.

REFERENCES

[1] Alonso, G., Agrawal, D., Abbadi, E., Kamath, M., Gunthor, R., Mohan, C. (1996), Advanced Transaction Models in Workflow Contexts, Proceedings of the 12th International Conference on Data Engineering, New Orleans, Louisiana, USA, 574-581.

[2] Bacon, D. (1991), Transparent Recovery in Distributed Systems, ACM Operating Systems Review, 91-94.

[3] Bernstein, P., Hadzilacos, V. and Goodman, N. (1987), Concurrency Control and Recovery in Database Systems, Addison-Wesley.

[4] Garcia-Molina, H., Ullman, J. and Widom, J. (2002), Database Systems, Prentice Hall.

[5] Garcia-Molina, H. and Salem, K. (1987), Sagas. Proc. ACM SIGMOD Int. Conf. on Management of Data, 249-259.

[6] Georgakopoulos, D. (1991), Multidatabase Recoverability and Recovery. 1st Int'l Workshop on Interoperability in Multidatabase Systems, 348-355.

[7] Georgakopoulos, D., Hornick, M. and Sheth, A. (1995), An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases, 3(2), 119-153.

[8] Gorbenko, A., Kharchenko, V., Popov, P., Romanovsky, A., Boyarchuk, A. (2004), Development of Dependable Web Services out of Undependable Web Components, University of Newcastle, School of Computer Science, Technical Report, 4-7.

[9] Gray, J. (1979), Notes on Data Base Operating Systems. Operating Systems: An Advanced Course. Springer Verlag, 393-481.

[10] Korth, H., Levy, E., and Silberschatz, A. (1990), A Formal Approach to Recovery by Compensating Transactions. Proc. of the 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia, 95-106.

[11] Krishnakumar, N. and Sheth, A. (1995), Managing Heterogeneous Multi-system Tasks to Support Enterprise-wide Operations. Distributed and Parallel Databases, 3(2), 155-186.

[12] Lee, P. and Anderson, T. (1990), Fault Tolerance Principles and Practice. Dependable Computing and Fault-Tolerant Systems. Springer-Verlag.

[13] Leymann, F., Güntzel, K. (2003), The Business Grid: Providing Transactional Business Processes via Grid Services, Proceedings of the International Conference on Service Oriented Computing (ICSOC), 256-270.

[14] Lomet, D. and Tuttle, M. (1995), Redo Recovery after System Crashes, Proc. of the 21st International Conference on Very Large DataBases (VLDB), 457-468.

[15] Lomet, D. and Tuttle, M. (1999), Logical Logging to Extend Recovery to New Domains, Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data, 73-84.

[16] Microsoft, BEA and IBM (2005), Web Services Atomic Transaction. ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf

[17] Microsoft, BEA and IBM (2005), Web Services Business Activity. ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf

[18] Microsoft, BEA and IBM (2005), Business Process Execution Language. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

[19] Microsoft, BEA and IBM (2005), Web Services Coordination, ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf

[20] Mikalsen, T., Tai, S., Rouvellou, I. (2002), Transactional Attitudes: Reliable Composition of Autonomous Web Services, International Conference on Dependable Systems and Networks.

[21] Miller, J., Sheth, A., Kochut, K. and Luo, Z. (1999), Recovery Issues in Web-Based Workflow, Proceedings of the 12th International Conference on Computer Applications in Industry and Engineering (CAINE-99), Atlanta, Georgia, 101-105.

[22] Moss, E. (1985), Nested Transactions, Cambridge, Mass. MIT Press.

[23] OASIS (2004), Business Transaction Protocol Committee Specification. 1-22. http://www.oasis-open.org/committees/business-transactions/

[24] OASIS (2005), Web Services Composite Application Framework. http://www.oasis-open.org/committees/ws-caf/

[25] Ozsu, M. and Valduriez, P. (1999), Principles of Distributed Database Systems, 2nd Edition. Prentice Hall.

[26] Papazoglou, M. (2003), Web Services and Business Transactions. World Wide Web: Internet and Web Information Systems, Tilburg University.

[27] Rusinkiewicz, M. and Sheth, A. (1995), Specification and Execution of Transactional Workflows, Modern Database Systems: The Object Model, Interoperability, and Beyond. Addison-Wesley, 592-620.

[28] Rusinkiewicz, M., Elmagarmid, A., Leu, Y., Litwin, W. (1990), Extending the Transaction Model to Capture more Meaning, SIGMOD Record, 459.

[29] Salzberg, B and Tombroff, D. (1996), DSDT: Durable Scripts Containing Database Transactions, IEEE International Conference on Data Engineering, 624-633.

[30] Samaras, G. et al. (1993): Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. Proc. of the 9th Conference on Data Engineering, Vienna, Austria, 520-529.

[31] Sauter, P. and Melzer, I. (2005), A Comparison of WS-BusinessActivity and BPEL4WS Long-Running Transaction, Proc. of Kommunikation in Verteilten Systemen (KIVS).

[32] Sheth, A., Rusinkiewicz, M. (1993), On Transactional Workflows, Bulletin of the Technical Committee on Data Engineering, 16(2).

[33] Sheth, A., Rusinkiewicz, M., Karabatis, G. (1992), Using Polytransactions to Manage Interdependent Data, Database Transaction Models for Advanced Applications, Morgan-Kaufmann, 555-581.

[34] Tai, S., Khalaf, R., and Mikalsen, T. (2004), Composition of Coordinated Web Services, Proceedings of the 5th ACM/IFIP/USENIX Intl. Conf. on Middleware. Toronto, Canada, 294-310.

[35] Tang, J., Veijalainen, J. (1995), Transaction-oriented Work-flow Concepts in Inter-organizational Environments, Proc. of the 4th Intl. Conf. on Information and Knowledge Management, Baltimore, 250-259.

[36] Tartanoglu, F., Issarny, V., Romanovsky, A., Levy, N. (2003), Coordinated Forward Error Recovery for Composite Web Services. Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS), Florence, Italy, 167-176.

[37] Verma, K., Gomadam, K., Lathem, J., Sheth, A. and Miller, J. (2006), Semantics enabled Dynamic Process Configuration, LSDIS Technical Report.

[38] Verma, K., Sheth., A. (2005), Autonomic Web Processes. In Proceedings of the Third International Conference on Service-oriented Computing (ICSOC), - Vision Paper, 2005. Lecture Notes in Computer Science (LNCS 3826), Springer Verlag. B. Benatallah, F. Casati, P. Traverso editors, 1-11

[39] Waechter, H. and Reuter, A. (1992), The ConTract Model, Database Transaction Models for Advanced Applications, 7(4), 219-263.

[40] Wirz, B. and Nett, E. (1993), A Generic Log-Service Supporting Fast Recovery in Distributed Fault-Tolerant Systems, IEEE Workshop on Advances in Parallel and Distributed Systems (APADS), Princeton, New Jersey.

[41] Worah, D. and Sheth, A. (1997), Transactions in Transactional Workflows, Advanced Transaction Models and Architectures, Kluwer Academic Publishers, 13-27.

[42] X/Open Data Management (1993), SQL Call Level Interface, X/Open Company, Ltd.

APPENDIX A: OPENWS-TRANSACTION INSTALLATION GUIDE

REQUIREMENTS

- JDK 1.4, available at http://java.sun.com/j2se/1.4.2/

  NOTE: Because of third-party libraries, the framework does not yet support JDK 1.5.
- Apache Tomcat 5.0. In subsequent pages, $CATALINA_HOME represents Tomcat's installation directory.
- BPEL-compliant process execution engine. The demo includes sample processes for easy deployment into ActiveBPEL, an open-source BPEL implementation.
- PostgreSQL and MySQL, database back-ends for the sample Web services.
- JDBC drivers for the above databases. They can be obtained from

  http://jdbc.postgresql.org/download.html and

  http://dev.mysql.com/downloads/connector/j/3.1.html

  Because they are needed by Tomcat's connection pool, place them in

  $CATALINA_HOME/common/lib.
- Apache Ant, used to deploy the sample BPEL processes.
- Optionally, Log4J's Chainsaw, a graphical interface used to monitor transaction progress.

INSTALLATION

1. Download the latest framework distribution from the LSDIS project page:

   http://lsdis.cs.uga.edu/projects/meteor-s/modules/OpenWSTransaction

2. Extract the contents into a location that we will call $WSTX_HOME from this point on.

3. Copy $WSTX_HOME/webapps/wstx.war into Tomcat's webapps folder.

4. Copy $WSTX_HOME/webapps/wstx-client.war into Tomcat's webapps folder.

5. Download and install your favorite BPEL implementation. For ActiveBPEL, visit http://www.activebpel.org. This document assumes ActiveBPEL as your process engine.

6. Deploy the sample business process:

   ```
   cd $WSTX_HOME/bpel/wsat
   ant deploy-bpel
   cd $WSTX_HOME/bpel/wsba
   ant deploy-bpel
   ```

7. Install sample database schemas:

   a. MySQL

      Create the database, user and schema from $WSTX_HOME/dbms/mysql/schema.sql:

      ```
      mysql -u root -p < schema.sql
      ```

      Note: This *root* user is not the operating system user but rather a special database user

   b. PostgreSQL:

      Create the database, user and schema from $WSTX_HOME/dbms/pgsql/schema.sql:

      ```
      createdb wstx -U postgres -W
      createlang plpgsql -d wstx -U postgres -W
      psql -U postgres -W wstx -f schema.sql
      ```

8. Download and run Log4J Chainsaw. The easiest way to achieve this is to use the Java WebStart feature, which is normally available with any JDK or JRE installation.

   Visit http://logging.apache.org/log4j/docs/chainsaw.html for details.

APPENDIX B: OPENWS-TRANSACTION USERS GUIDE

ABOUT THE SAMPLE PROCESS

The included sample application simulates the popular travel arrangements business process: A travel agent initiates a transactional process involving an airline, a hotel reservation system and a banking system, all available as Web services. For the process to succeed, all participants have to carry out their operations successfully, otherwise the transaction is aborted.

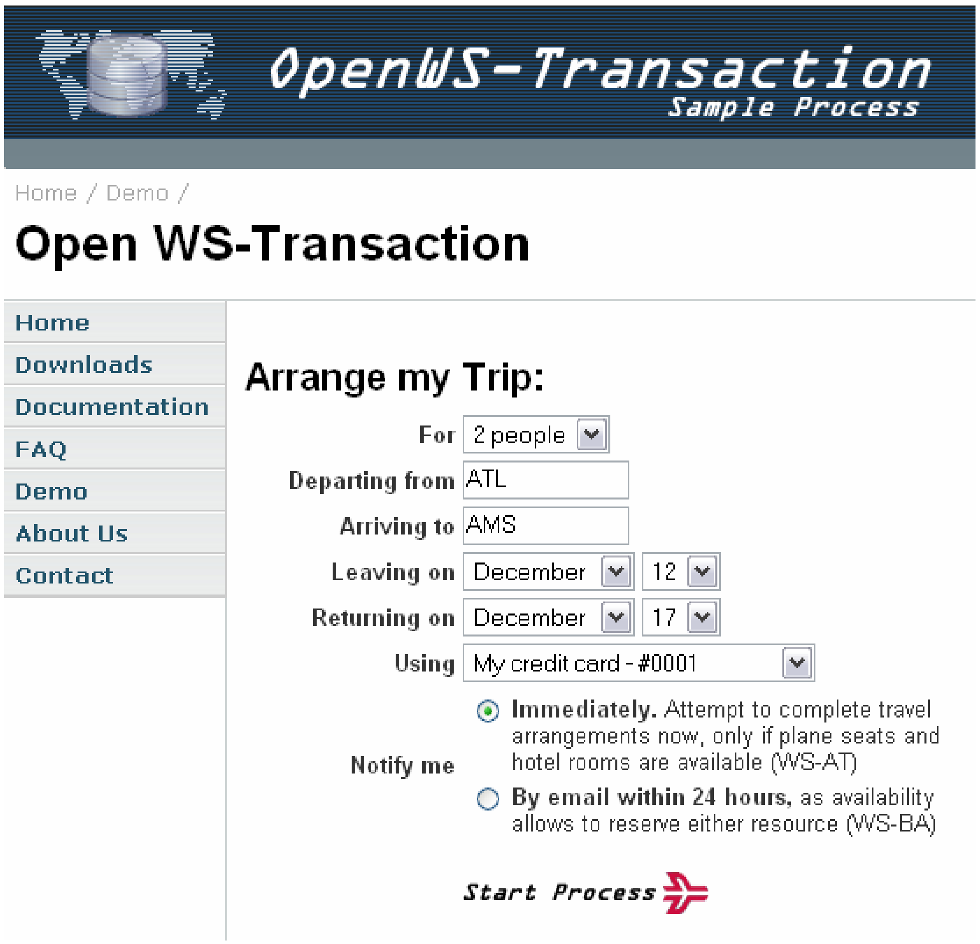
Failures can occur at any time in the process, however, the framework can bring participants back to their original state. For example, once the decision is made to commit the transaction, some database back end can go down, undoing any pending operations (under WS-AtomicTransaction). Once the system comes back up, the framework will ask that participant to recover, allowing it to complete its part of the work as originally expected.

In another scenario, the server hosting the transaction coordinator may crash, loosing all state information about pending transactions. Once the server becomes available, it can be recovered by gathering information from all participants (and recover them in turn, if necessary) in order to resume operation.

RUNNING THE SAMPLE PROCESS

1. Start the Chainsaw log viewer. If you have Java WebStart, it can be started by simply clicking the appropriate link on the URL provided in Appendix B. Once it starts, configure a SocketReceiver on port 4560 to listen incoming logging events.

2. Go to http://localhost:8080/wstx-client/ and select an itinerary of your choice. The “Start Process” button starts the sample process which, under normal circumstances will succeed or fail depending on the data stored at each participants' local database.



Open WS-Transaction

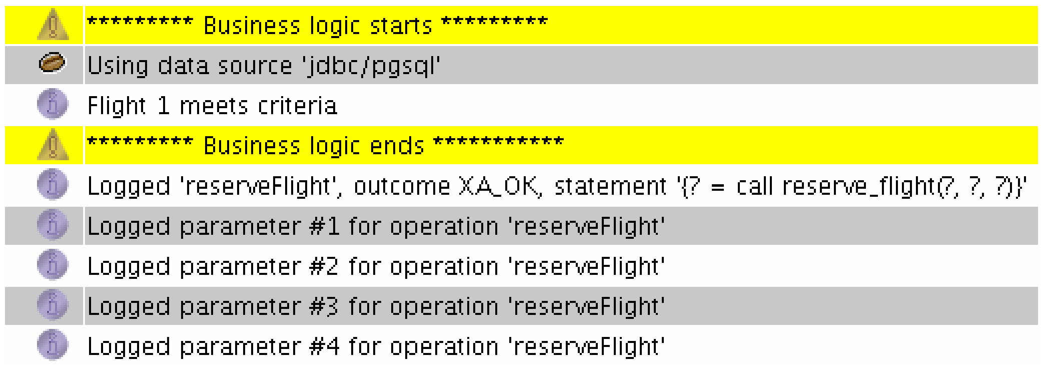
Arrange my Trip:

For: 2 people
Departing from: ATL
Arriving to: AMS
Leaving on: December 12
Returning on: December 17
Using: My credit card - #0001
Notify me: [✓ **Immediately.** Attempt to complete travel arrangements now, only if plane seats and hotel rooms are available (WS-AT)] **By email within 24 hours,** as availability allows to reserve either resource (WS-BA)

Figure 17: Web client used to initiate the sample process

3. Monitor underlying database activity using the psql and mysql command-line tools or using a GUI interface such as QuantumDB eclipse plugin. For WS-AT transactions, check pending transactions using:

Postgresql:	select * from pg_locks
MySQL:	show innodb status



```
********* Business logic starts *********
Using data source 'jdbc/pgsql'
Flight 1 meets criteria
********* Business logic ends ***********
Logged 'reserveFlight', outcome XA_OK, statement '{? = call reserve_flight(?, ?, ?)}'
Logged parameter #1 for operation 'reserveFlight'
Logged parameter #2 for operation 'reserveFlight'
Logged parameter #3 for operation 'reserveFlight'
Logged parameter #4 for operation 'reserveFlight'
```

Figure 18: Logging of database activity following a successful operation

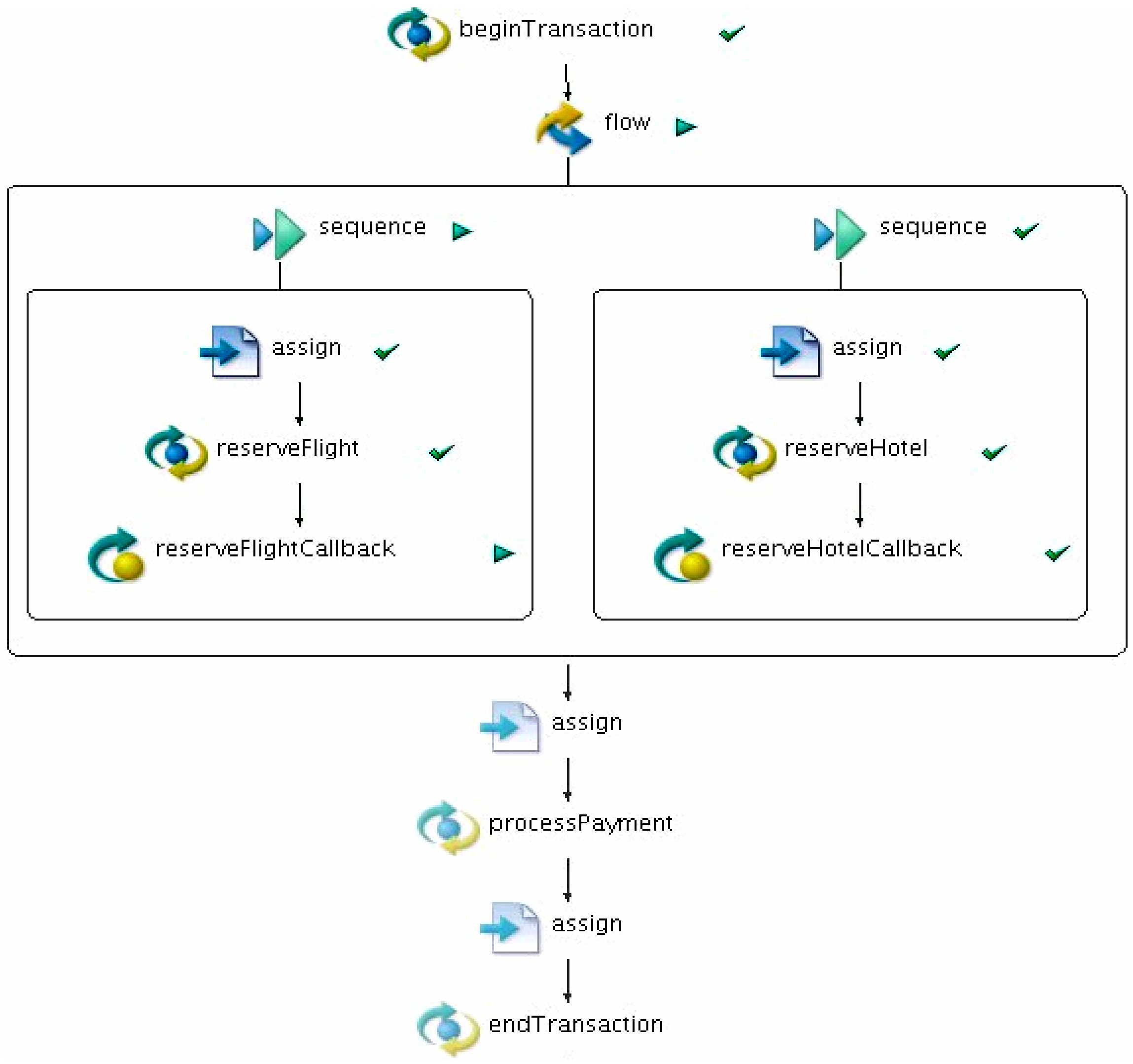
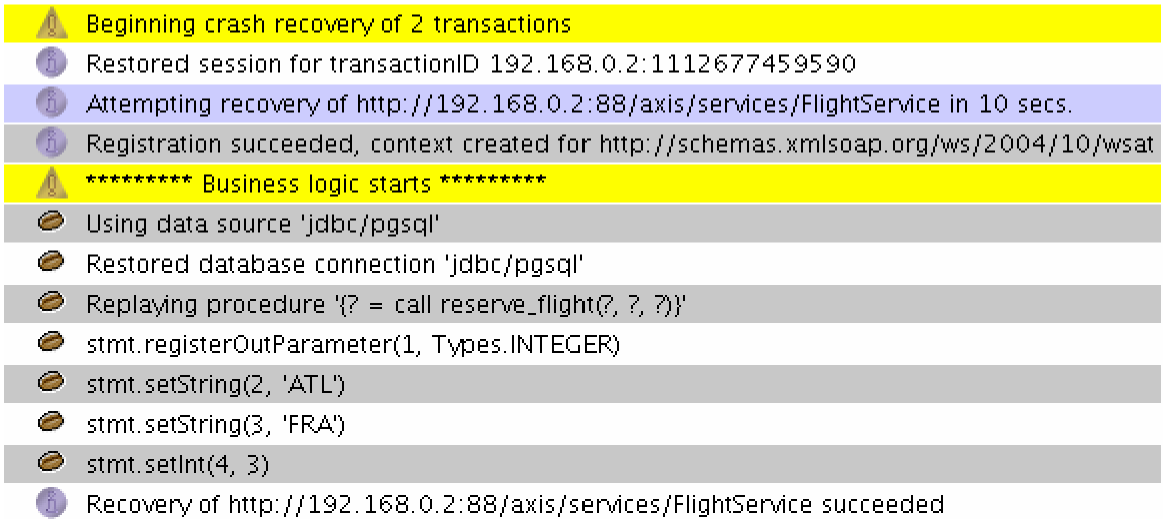


Figure 19: Monitoring the sample transactional business process
Created with ActiveBPEL. Not for Commercial Use

4. Simulate failures while in the middle of a transaction. For example:

- Shut down any database after the coordinator has decided to commit. At the end of the transaction protocol, the coordinator will ask the affected participant to recover, causing it to restore its state and replay any pending database activity.
- Shut down Tomcat. Once it restarts, the framework will automatically run the coordinator's recovery procedure and restore the state of any pending transactions.



```
Beginning crash recovery of 2 transactions
Restored session for transactionID 192.168.0.2:1112677459590
Attempting recovery of http://192.168.0.2:88/axis/services/FlightService in 10 secs.
Registration succeeded, context created for http://schemas.xmlsoap.org/ws/2004/10/wsat
********* Business logic starts *********
Using data source 'jdbc/pgsql'
Restored database connection 'jdbc/pgsql'
Replaying procedure '{? = call reserve_flight(?, ?, ?)}'
stmt.registerOutParameter(1, Types.INTEGER)
stmt.setString(2, 'ATL')
stmt.setString(3, 'FRA')
stmt.setInt(4, 3)
Recovery of http://192.168.0.2:88/axis/services/FlightService succeeded
```

Figure 20: Coordinator recovery upon server restart

- Increase the delay of interations, forcing transactions to time out. You can do so by changing $CATALINA_HOME/webapps/wstx/WEB-INF/classes/wstx.properties as follows:

coordinator_delay=60

CREATING TRANSACTIONAL BUSINESS PROCESSES

Delivering transactional capabilities to an existing business process involves the following tasks:

1. Making Available the Coordinator Service

   This is accomplished as soon as the framework is installed. To verify the coordinator's availability, users can browse to http://localhost:8080/wstx/services/Coordinator, which should respond as a valid Web service URL.

2. Delivering Transactional Awareness to Participating Services

   The framework attempts to minimize the number of code changes required to convert conventional Web services into transaction-aware participants. Consequently, only a few lines need to be changed, which we explain below:

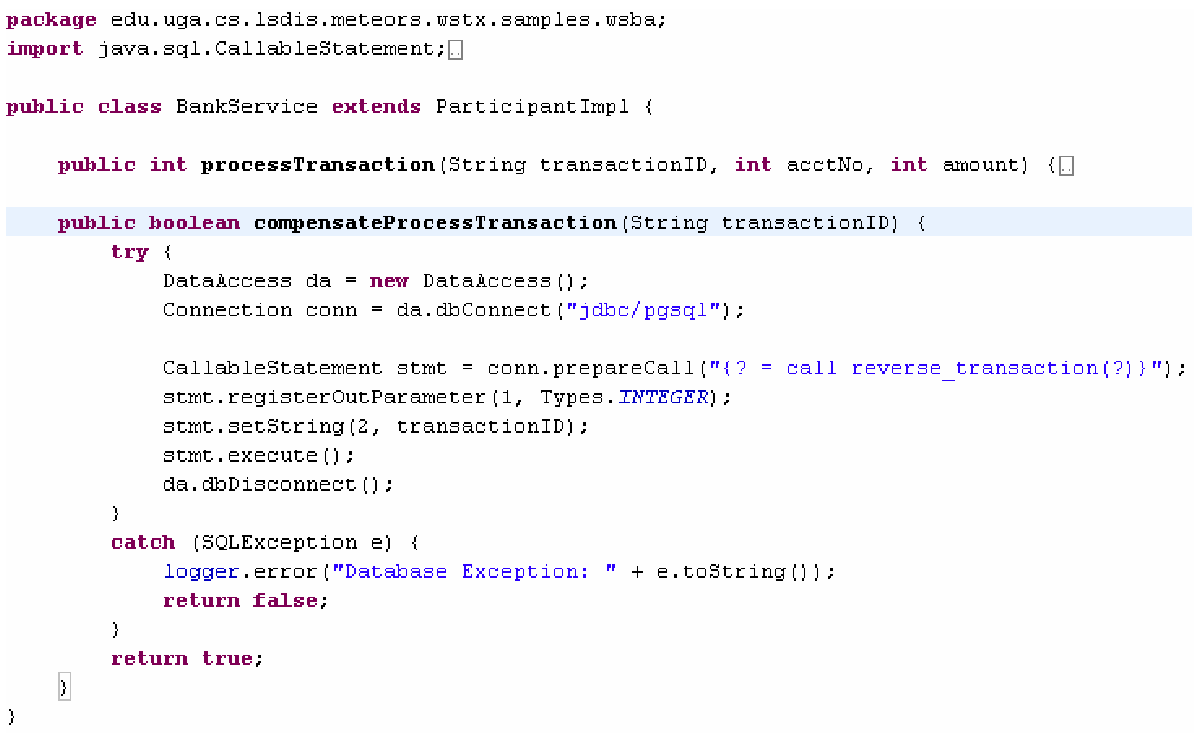
   a. Extend the Participant framework class. This step causes the service implementation to inherit most of the functional requirements any transactional Web service must meet.

   b. If the process is to follow the WS-AT coordination type, obtain a database connection from the participant entity, rather by directly accessing a data source. This allows the framework to intercept and log database activity in case it has to be replayed due to system failures.

      NOTE: This step is not necessary under the WS-BA coordination type, as partial operations are always immediately persisted.

   c. Notify the framework about the logical outcome of the service's activity. Following the completion of an activity, the service must report its successful completion, otherwise, the Participant entity will take a pessimistic guess and vote to abort the transaction.

For services taking part in long-running WS-BA transactions, it is necessary to define compensating actions, which will be invoked by the coordinator service if the transaction has to be aborted. Figure 21 shows an excerpt from the supplied WS-BA example service.

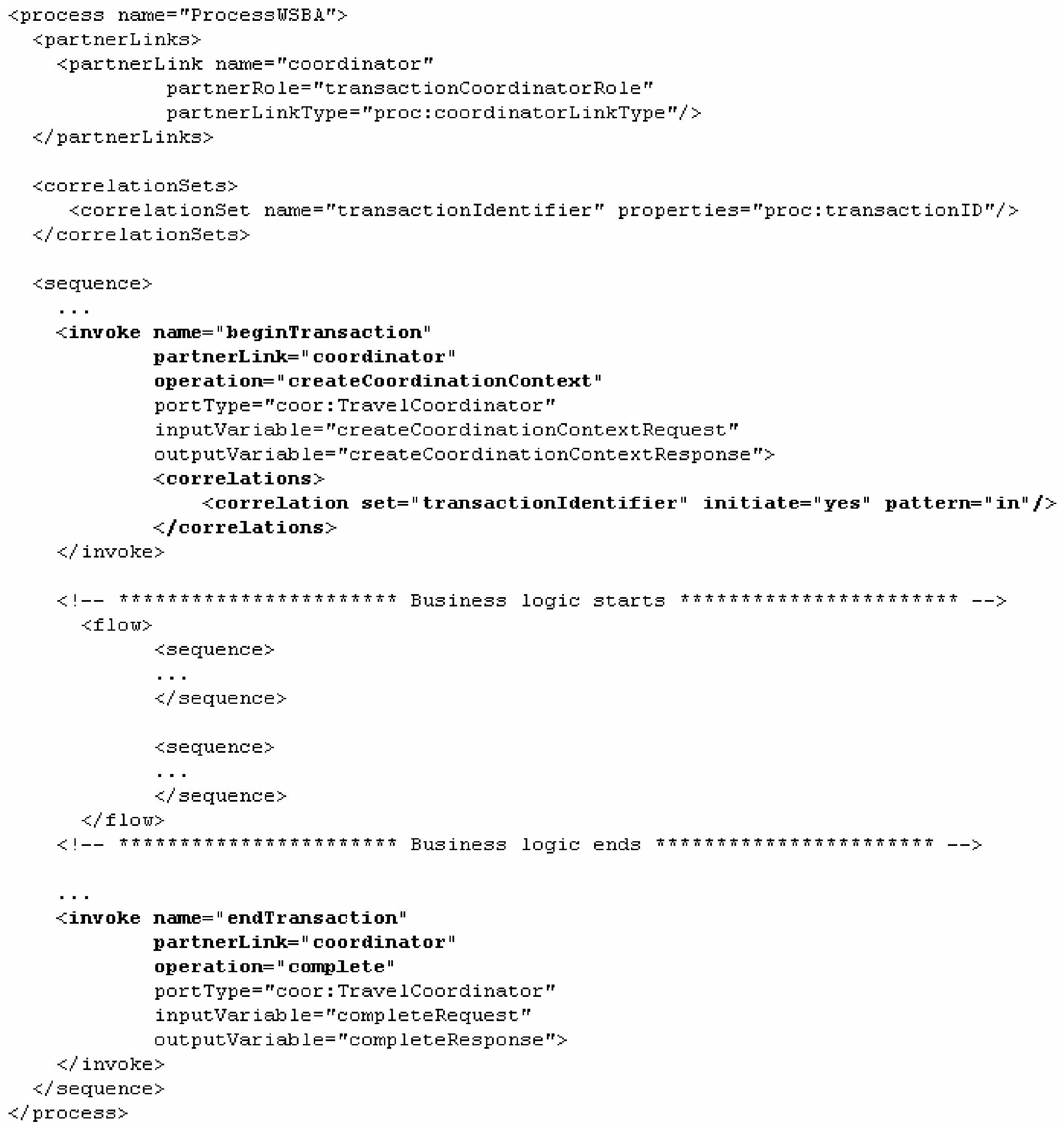


```
package edu.uga.cs.lsdis.meteors.wstx.samples.wsba;
import java.sql.CallableStatement;

public class BankService extends ParticipantImpl {

    public int processTransaction(String transactionID, int acctNo, int amount) {

    public boolean compensateProcessTransaction(String transactionID) {
        try {
            DataAccess da = new DataAccess();
            Connection conn = da.dbConnect("jdbc/pgsql");

            CallableStatement stmt = conn.prepareCall("{? = call reverse_transaction(?)}");
            stmt.registerOutParameter(1, Types.INTEGER);
            stmt.setString(2, transactionID);
            stmt.execute();
            da.dbDisconnect();
        }
        catch (SQLException e) {
            logger.error("Database Exception: " + e.toString());
            return false;
        }
        return true;
    }
}
```

Figure 21: Implementation of a user-defined compensating operation

3. Defining a Transactional Scope in the Business Process

   All service invocations requiring transactional behavior must be enclosed within a transactional scope, as illustrated in Figure 22.



```
<process name="ProcessWSBA">
  <partnerLinks>
    <partnerLink name="coordinator"
              partnerRole="transactionCoordinatorRole"
              partnerLinkType="proc:coordinatorLinkType"/>
  </partnerLinks>

  <correlationSets>
     <correlationSet name="transactionIdentifier" properties="proc:transactionID"/>
  </correlationSets>

  <sequence>
    ...
    <invoke name="beginTransaction"
            partnerLink="coordinator"
            operation="createCoordinationContext"
            portType="coor:TravelCoordinator"
            inputVariable="createCoordinationContextRequest"
            outputVariable="createCoordinationContextResponse">
            <correlations>
                <correlation set="transactionIdentifier" initiate="yes" pattern="in"/>
            </correlations>
    </invoke>

    <!-- *********************** Business logic starts *********************** -->
      <flow>
            <sequence>
            ...
            </sequence>

            <sequence>
            ...
            </sequence>
      </flow>
    <!-- *********************** Business logic ends *********************** -->

    ...
    <invoke name="endTransaction"
            partnerLink="coordinator"
            operation="complete"
            portType="coor:TravelCoordinator"
            inputVariable="completeRequest"
            outputVariable="completeResponse">
    </invoke>
  </sequence>
</process>
```

Figure 22: A BPEL process definition with a transactional scope

APPENDIX C: PERFORMANCE EVALUATION

The experimental results made clearly evident the overhead caused by coordination, which is well beyond that required to support fault tolerance. To a large degree, this overhead results from the increased complexity of interactions as prescribed by coordination protocols, rather than by a particular design or implementation decision.

Under the most straightforward execution path (e.g., successful invocation of all services, one operation per participant, etc.), a non-coordinated process requires only one interaction with each service. Taking into account that each interaction requires two messages (i.e., request and response) and with n equal to the number of participants, we have:

$$\textit{exchanged messages} = 2n$$

For comparison, using WS-AT coordination, the protocol prescribes the intervention of an additional service (i.e., the coordinator) and the interactions described in Section 4.3, which result in:

$$\textit{exchanged messages} = 2n + 4 + 2n + n + 2n = 7n + 4$$

The first term corresponds to messages between the business process and its participants. The second is the constant number of messages required to delimit the transactional scope. The third term is the number of messages required to register all participants with the coordinator. The fourth corresponds to the one-way unsolicited vote sent by each participant to the coordinator. Finally, the fifth term reflects the number of messages exchanged between the coordinator and its participants in order to carry the last phase of the transaction protocol. Thus, the increase in exchanged messages due to coordination can be approximated by the ratio between the two:

$$\textit{increase in exchanged messages} = (7n+4)/2n \approx 3.5$$

Therefore, a process coordinated under WS-AT requires approximately 3.5 times more message exchanges, representing a corresponding increase in network utilization and message processing overhead.

With regards to recovery, we measured its performance by averaging recovery times for a single transaction failing at different stages. The time required to recover from failures occurring shortly after context creation or participant registration is dominated by the time to analyze the log (nearly 200 ms in Figure 23), as they only involve restoring the coordination context. Then, recovery time grows proportional to the number of operations already performed, given the need to gather participant votes in order to recover the coordinator (this tradeoff that was clearly anticipated in favor of faster execution times under normal conditions). For WS-AT transactions, recovery is generally more time-consuming than WS-BA due to the time spent rebuilding and resubmitting database activity. The greatest time to recover (close to 3 seconds in Figure 16) was reached when all participants in the transactional scope had been executed, yet a decision had not been made about the transaction. In that case, a recovering coordinator must not only poll all participants (potentially asking some of them to recover), but also make a decision and carry out the transaction protocol. If the failure occurs once the decision has been recorded, for WS-AT the recovery time is progressively lower as finished participants increase; instead, for WS-BA this time almost negligible as no database activity has to be replayed (it would certainly be higher if compensating actions had to be applied at the end of the transaction protocol).

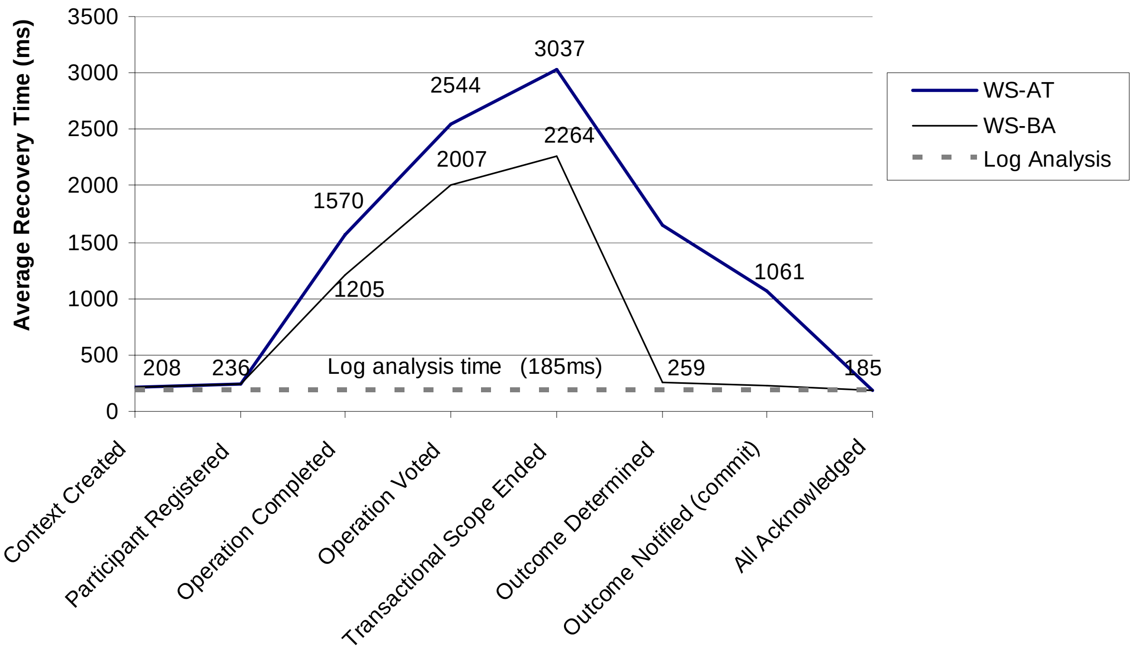


Figure 23: Recovery time relative to transaction status

With respect to the prototype’s log disk space consumption, every transaction required approximately 15 kilobytes (KB) of storage, distributed among the coordinator (4.9KB) and three participants (3.7 KB each). Since the prototype’s logging facility is based on an embedded database system, we have no direct control over optimizations at this level.